

Monday Oct. 7  
Lecture 8

Employees:

name	id	salary
alan	2	4500.34
mark	3	3450.67
tom	1	3450.67

List 1: tom alan mark  
List 2: alan tom mark

highest salary

Alternatively "smallest" object on the list

mark 3 alan 2 tom 1

Sorting based on id's

List 1 → tom alan mark

→ Sorting based on salaries and id's

List 2 → alan tom mark

emp smaller if id smaller

larger comes first

smaller comes first

# Comparable Employee: Version I

```
class Employee1 implements Comparable Employee1 {
    ... /* attributes, constructor, mutator similar to Employee */
    @Override
    public int compareTo(Employee1 e) { return this.id - e.id; }
}
```

generic parameter

alan.compareTo(mark);

2 "alan > mark" 3  
 tom.compareTo(alan)

"tom > alan" 1  
 return (e.id - this.id);

→ this.id - e.id  
 e.id - this.id

< 0

> 0

→ == 0

tom > alan > mark

2 0 1

```
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};
    assertEquals(expected, es);
}
```

mark < alan  
 alan < tom

alan.compareTo(mark); -1

alan < mark

tom.compareTo(alan); -1

1 2  
 tom < alan

tom < alan < mark

# Comparable Employee: Newton (2.1)

Double. compare (alan. salary, mark. salary);

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



Double

without args,  
alan will  
appear later  
than mark  
in the  
list.

```
@Test  
public void testComparableEmployees_2() {  
    /*  
     * CEmployee2 implements the Comparable interface.  
     * Method compareTo first compares salaries, then  
     * compares id's for employees with equal salaries.  
     */  
    CEmployee2 alan = new CEmployee2(2);  
    CEmployee2 mark = new CEmployee2(3);  
    CEmployee2 tom = new CEmployee2(1);  
    alan.setSalary(4500.34);  
    mark.setSalary(3450.67);  
    tom.setSalary(3450.67);  
    CEmployee2[] es = {alan, mark, tom};  
    Arrays.sort(es);  
    CEmployee2[] expected = {alan, tom, mark};  
    assertEquals(expected, es);  
}
```

alan < mark

# Comparable Employee: Version 2.2

```
class CEmployee2 implements Comparable<CEmployee2> {
    ... /* attributes, constructor, mutator similar to Employee */
    @Override
    public int compareTo(CEmployee2 other) {
        if (this.salary > other.salary) {
            return -1;
        }
        else if (this.salary < other.salary) {
            return 1;
        }
        else { /* equal salaries */
            return this.id - other.id;
        }
    }
}
```

Annotations and corrections:  
- Red 'X' marks on the first two lines of the compareTo method.  
- Red arrows pointing to 'this.salary' and 'other.salary' in the first two conditions.  
- Red circles around '-1' and '1' in the return statements.  
- Red underlines under 'other.salary' in both conditions.  
- Red squiggly underline under 'this.id - other.id' in the else block.  
- Red arrow pointing to 'this.id - other.id' with the text 'mark.id - tom.id' written below it.

larger salary  
↳ occur earlier in the sorted list  
↳ considered as "smaller"

$V > S$   
 $S > P$   
 $\Rightarrow V > P \times$

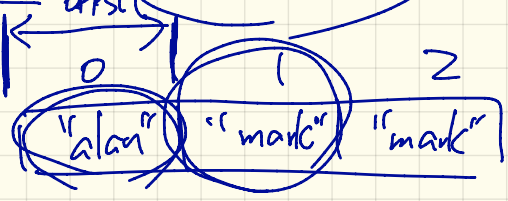
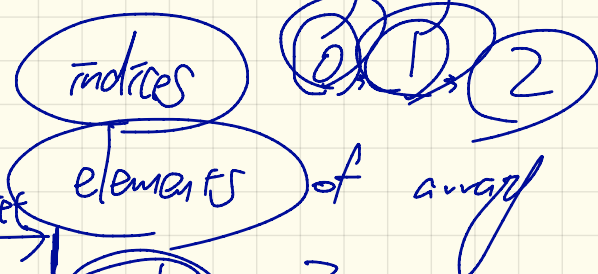
alan.compareTo(mark); -1  
→ "alan < mark"  
alan.compareTo(tom); -1  
→ "alan < tom"  
mark.compareTo(tom); 0  
this other  
→ "mark > tom"

```
@Test
public void testComparableEmployees_2() {
    /*
     * CEmployee2 implements the Comparable interface.
     * Method compareTo first compares salaries, then
     * compares id's for employees with equal salaries.
     */
    CEmployee2 alan = new CEmployee2(2);
    CEmployee2 mark = new CEmployee2(3);
    CEmployee2 tom = new CEmployee2(1);
    alan.setSalary(4500.3);
    mark.setSalary(450.6);
    tom.setSalary(450.6);
    CEmployee2[] es = {alan, mark, tom};
    Arrays.sort(es);
    CEmployee2[] expected = {alan, tom, mark};
    assertEquals(expected, es);
}
```

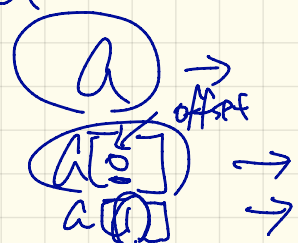
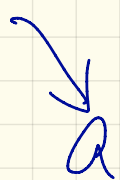
Red arrow pointing to the expected array in the assertEquals call: `assertEquals(expected, es);`

String[] names = {"alan", "mark", "mark"};

map → entries ✓  
 ↳ keys  
 ↳ values



2031

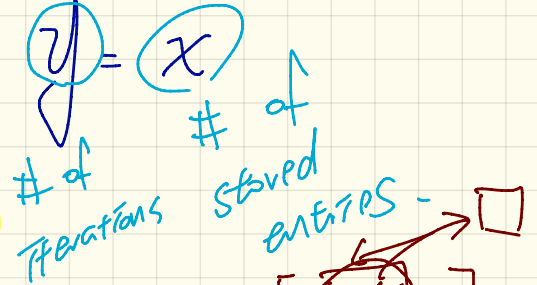


beginning address of array  
 go directly  
 go to address with 1 unit of offset.

# Implementing a Map using an Array

ENTRY	
(SEARCH) KEY	VALUE
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

Worst Case -

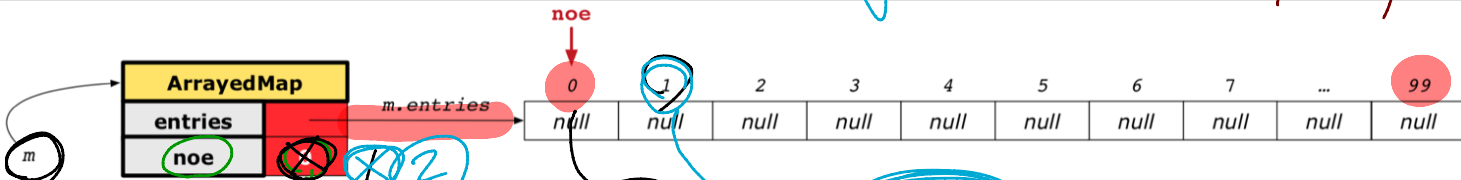


Entry	
key	
value	

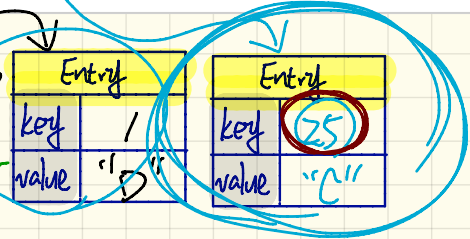
m.entries[25]

m.entries[0] =  
m.get(25)

a key but not the correct index to look up.

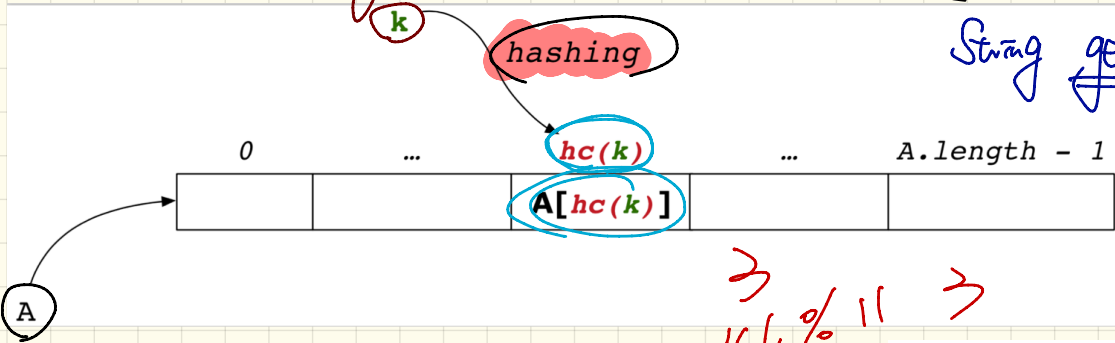


- # of entries
- next available slot to store a new entry



Hashing  $\rightarrow$  m.get(1)  
 $\rightarrow$  m.get(3)  
 key is 3

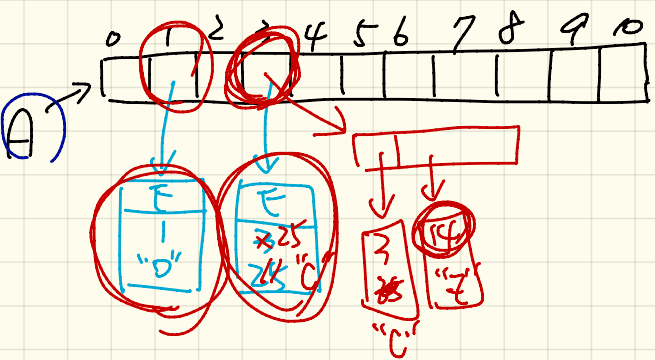
$A[\frac{0}{2}] \rightarrow$  efficient.



String get(int key) {  
 return  $A[key \% 11]$ ;  
 }  
 $1 \% 11 = 1$   
 $3 \% 11 = 3$

$3$   
 $14 \% 11 = 3$

Say. A.length is 11 and  
 $hc(k) = k \% 11$



	ENTRY	
hc(k)	(SEARCH) KEY	VALUE
1	1	D
3	25	C
	3	F
	14	Z
	6	A
	39	C
	7	Q