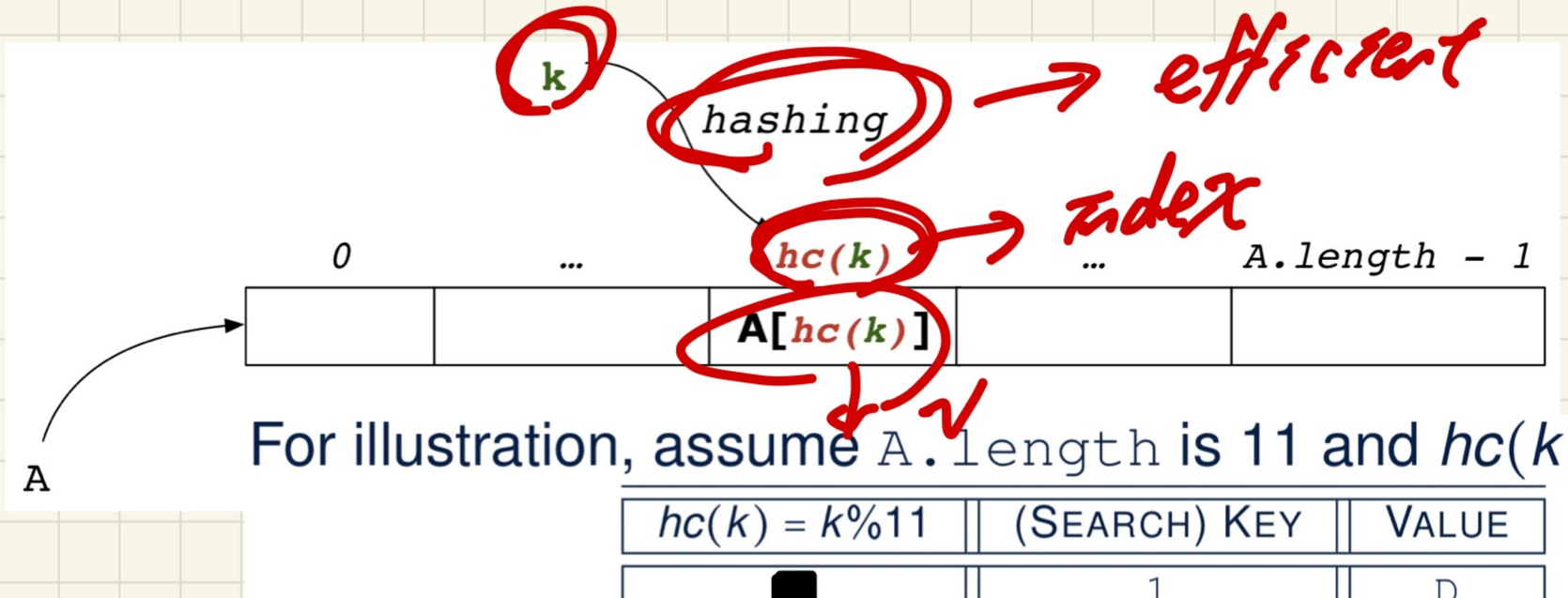
Lecture 11

Wednesday October 9

Implementing a Hash Table via Hashing



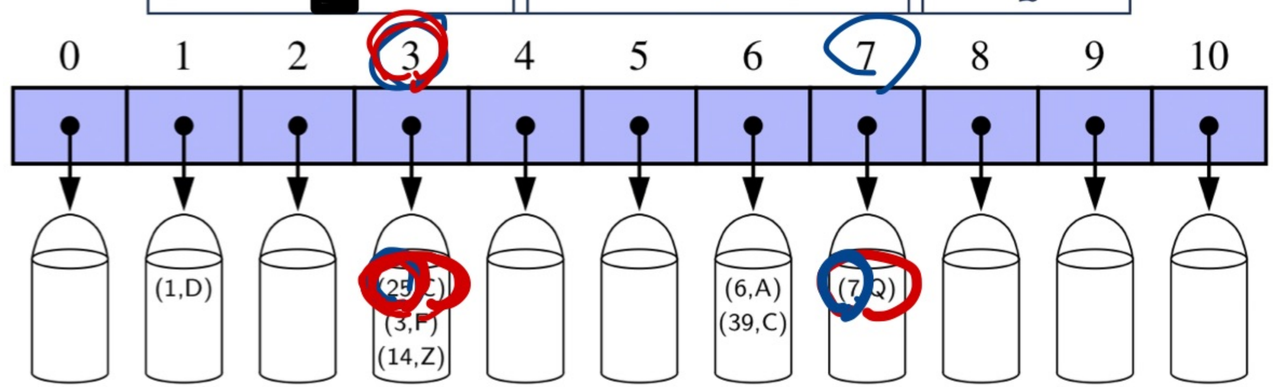
- Converting k to hc(k)
- Indexing into A[hc(k)]

For illustration, assume `A.length` is 11 and $hc(k) = k\%11$.

$hc(k) = k\%11$	(SEARCH) KEY	VALUE
1	25	D
3	3	C
14	14	F
6	6	Z
39	39	A
7	7	C
		Q

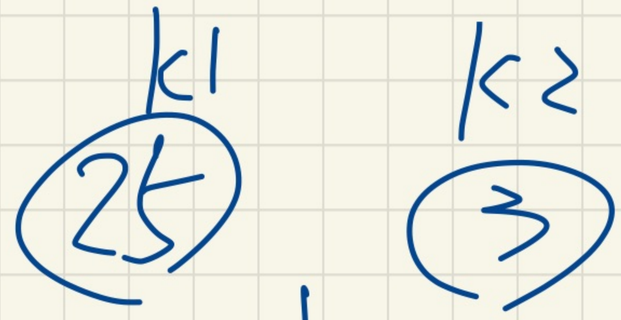
25 ≠ 7

3 ≠ 7



- k1.equals(k2) hc(k) = k % 15

- hc(k1) == hc(k2)



- hc(k1) != hc(k2)

25 7

hc(k1) == hc(k2)

hc(k) = k % 11

k1.equals(k2)

k1 = 25

k2 = 3

3

k1.equals(k2)

25

25]

hc & function

found.

Contract of a Hash Code Function

- Principle of defining a hash function *hc*:

$$k1.equals(k2) \Rightarrow hc(k1) == hc(k2)$$

  Equal keys always have the same hash code.
- Equivalently, according to contrapositive:

$$hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$$

$p \Rightarrow q \equiv$

$\neg q \Rightarrow \neg p$

"It rains" ⇒ "I bring um..."

¬ "I bring um..." ⇒ ¬ "It rains"

- What if k1.equals(k2) is false?
- What if hc(k1) == hc(k2) is true?

Overridding/Redefining hashCode() from Object

```
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.hashCode() == other.hashCode();
13   } }
```

- Principle of defining a hash function ***hc***:

  k1.*equals*(***k2***) ⇒ ***hc***(***k1***) == ***hc***(***k2***)

  Equal keys always have the same hash code.
- Equivalently, according to contrapositive:

  hc(***k1***) ≠ ***hc***(***k2***) ⇒ ¬***k1***.*equals*(***k2***)

Q: Can we replace Line 12 by:

```
return this.hashCode() == other.hashCode();
```

k1

1

hc(k1)

1

k2

12

hc(12)

1

new IntegerKey(1)

m.get(4)

1

equals true

bucket array:

add (1, "A")

add (12, "B")

for (every entry e in bucket array) {

"A"

"B"

12

equal keys. IntegerKey

3 objects

```
IK ik1 = new Ik(3);
IK ik2 = new Ik(3);
```

- Principle of defining a hash function ***hc***:

$$k1.equals(k2) \Rightarrow hc(k1) == hc(k2)$$

  Equal keys always have the same hash code.
- Equivalently, according to contrapositive:

$$hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$$

ik1. equals (ik2)

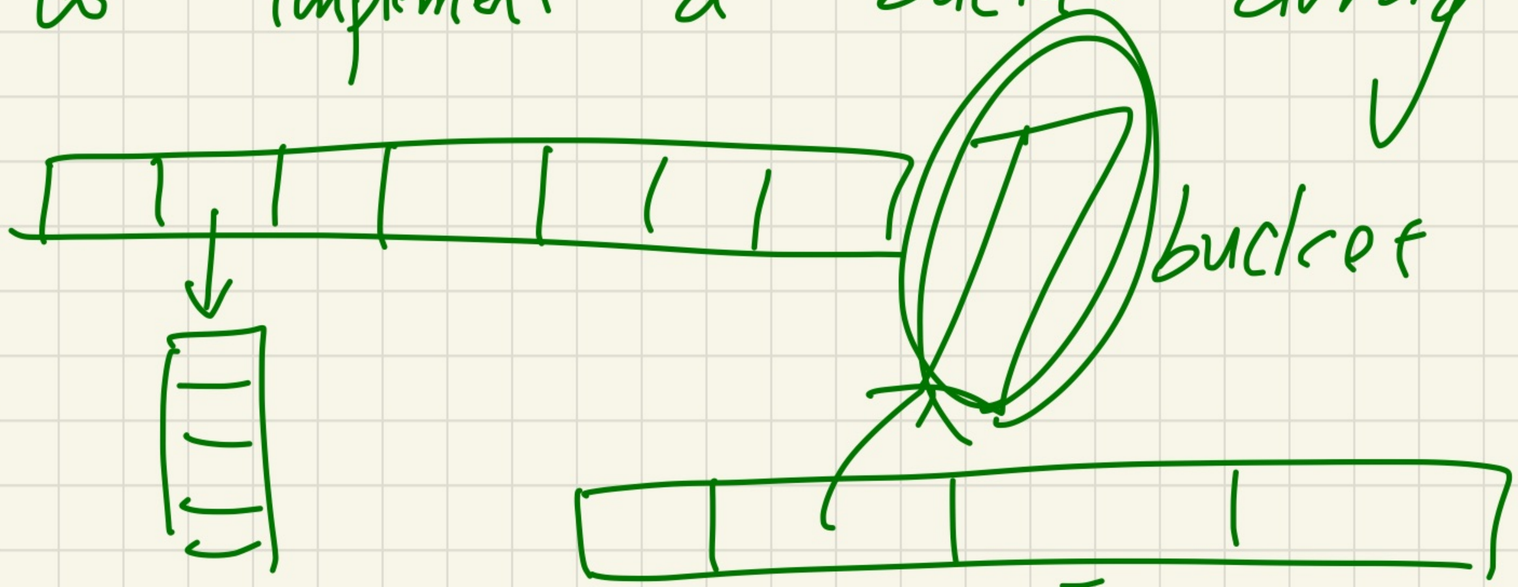
ik1. hc() == ik2. hc()

F

ik1 != ik2

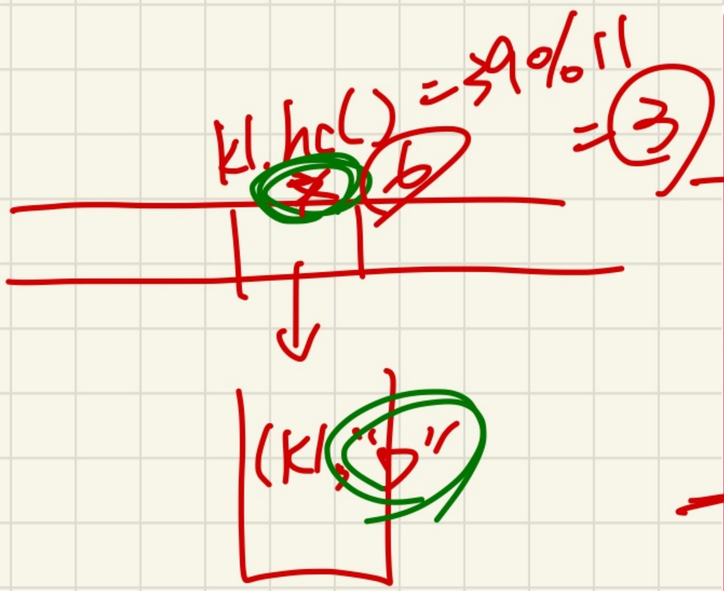
How to implement a bucket array?

1.



2. ArrayList<ArrayList<>> Entry.

Testing HashTable using Overridden/Redefined hashCode()



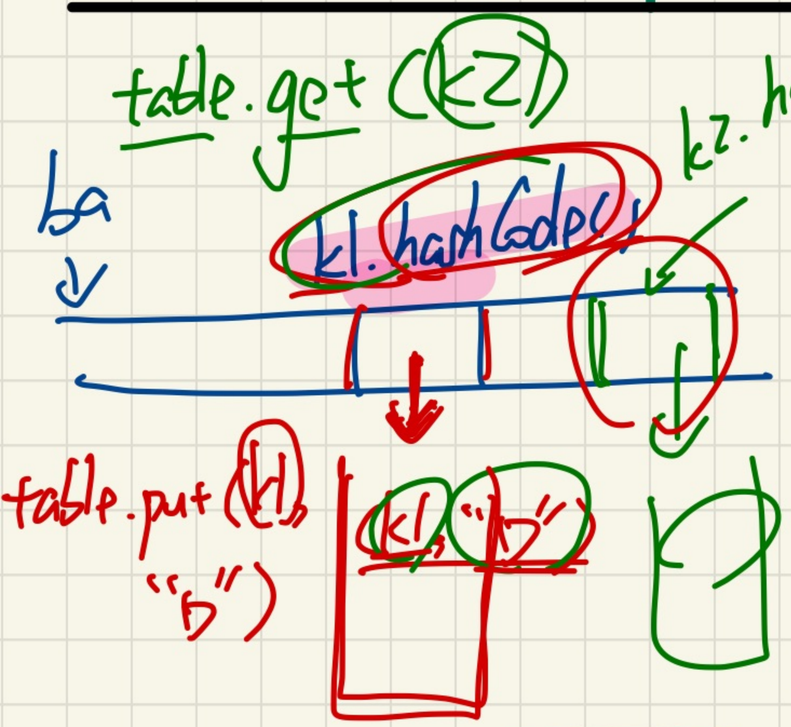

```
@Test
public void testHashTable() {
  Hashtable<IntegerKey, String> table = new Hashtable<>();
  IntegerKey k1 = new IntegerKey(39);
  IntegerKey k2 = new IntegerKey(39);
  assertTrue(k1.equals(k2));
  assertTrue(k1.hashCode() == k2.hashCode());
  table.put(k1, "D");
  assertTrue(table.get(k2).equals("D"));
}
```

table.put(k, v) → k.hashCode()

```
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   } }
```

table.get(k2) → k2.hc() = 11 = 3 ≠ b

Inconsistent equals and hashCode




```java
@Test
public void testDefaultHashFunction() {
 IntegerKey ik39_1 = new IntegerKey(39);
 IntegerKey ik39_2 = new IntegerKey(39);
  assertTrue(ik39_1.equals(ik39_2));
  assertTrue(ik39_1.hashCode() != ik39_2.hashCode()); }
@Test
public void testHashTable() {
  Hashtable<IntegerKey, String> table = new Hashtable<>();
 IntegerKey k1 = new IntegerKey(39);
 IntegerKey k2 = new IntegerKey(39);
  assertTrue(k1.equals(k2));
  assertTrue(k1.hashCode() != k2.hashCode());
  table.put(k1, "D");
  assertTrue(table.get(k2) == null); }
```

∵ k1 and k2 are different objects

```java
public class IntegerKey {
  private int k;
  public IntegerKey(int k) { this.k = k; }
  /* hashCode() inherited from Object NOT overridden. */
  @Override
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    IntegerKey other = (IntegerKey) obj;
    return this.k == other.k;
  }
}
```

→ default:
hash code
↳ based on address

Method Call: Callee vs. Caller

```
class A {
   ...
   void m(T param) {
     /* use of param */
   }
}
```

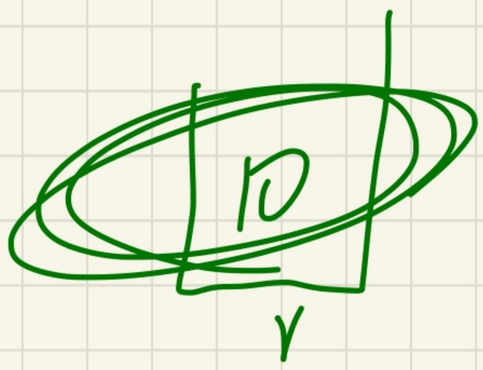
```
class B {
   ...
   void n(...){
     A co = new A();
     co.m(arg);
   }
}
```

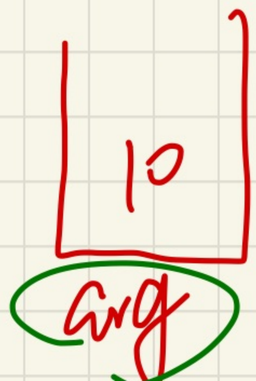
primitive or reference

Call by Value: Primitive Argument

```
class Circle {
   int radius;
   void setRadius(int r) {
      this.radius = r;
   }
}
```

```
class CircleUser {
   ...
   Circle c = new Circle();
   int arg = 10;
   c.setRadius(arg);
   }
}
```



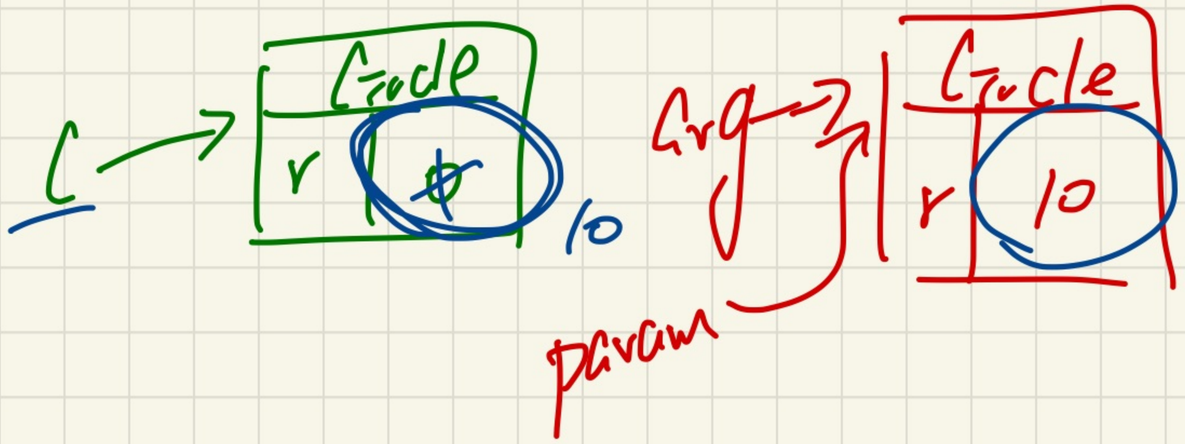


Call by Value: Reference Argument

```
class Circle {
  int radius;
  Circle() {}
  Circle(int r) {
    this.radius = r;
  }
  void setRadius(Circle param) {
    this.radius = param.radius;
  }
}
```

```
class CircleUser {
  ...
  Circle c = new Circle();
  Circle arg = new Circle(10);
  c.setRadius(arg);
  }
}
```

reference type



Call by Value: Re-Assigning Primitive Parameter

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByVal() {
3    Util u = new Util();
4    int i = 10;
5    assertTrue(i == 10);
6    u.reassignInt(i);
7    assertTrue(i == 10);
8  }
```