


# CS 10: Problem solving via Object Oriented Programming

Hashing

# Java provides us faster Sets and Maps using hashing instead of Trees

- Sets hold unique objects, Maps hold Key/Value pairs
- Map Keys are unique, but Values may be duplicated
- As we saw last class, using a Tree is a natural fit for implementing Sets and Maps
- Performance with a Tree is generally better than a List
- We can do better than Tree performance by using today's topic of discussion – hashing
- Java provides the HashSet and HashMap out-of-the-box that do a lot of the hard work for us

# Agenda

- 
1. Hashing
  2. Computing Hash functions
  3. Implementing Maps/Sets with hashing
  4. Handling collisions
    1. Chaining
    2. Open Addressing

# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)

Slots behind desk

00
01
02
03

•

•

•

98
99

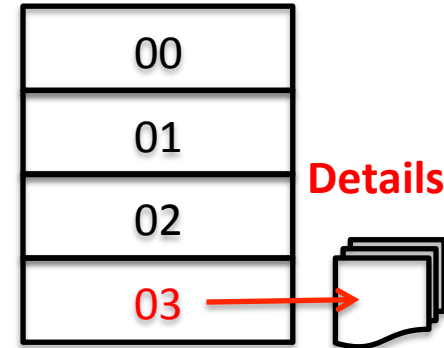
Fixed size  
table

# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)

Slots behind desk



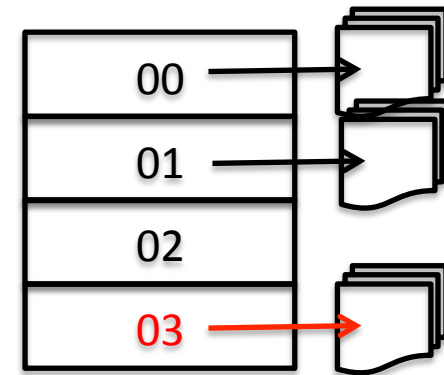
Fixed size  
table

# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone

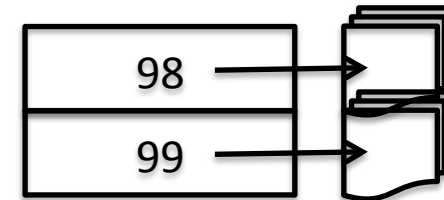
Slots behind desk



•

•

•



Fixed size  
table

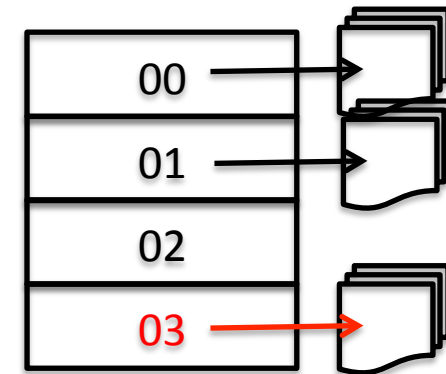
# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot

*Search only these orders, skip the rest*

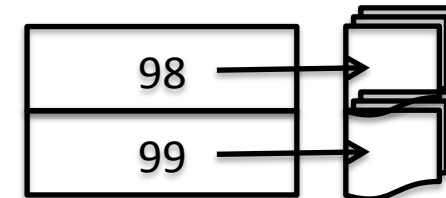
Slots behind desk



•

•

•



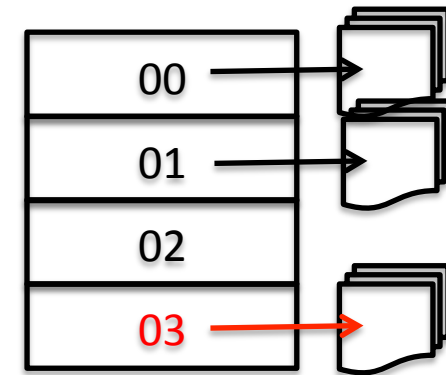
Fixed size  
table

# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot
- Splits set of (possibly) hundreds or thousands of orders into 100 slots of a few items each

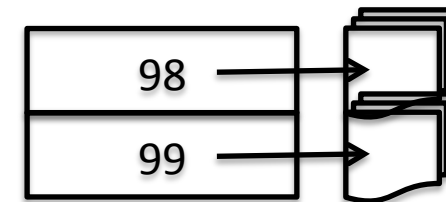
Slots behind desk



•

•

•



Fixed size  
table

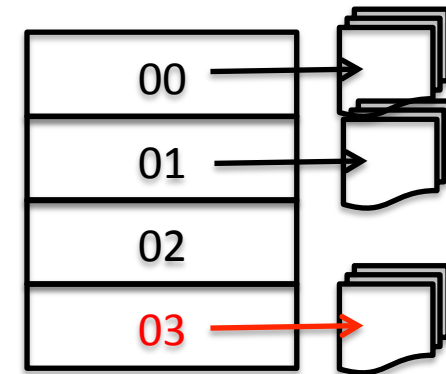


# The old Sears catalog orders illustrate how hashing works

## Sears store implementation of hash table

- Used to have 100 slots behind order desk, 0...99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot
- Splits set of (possibly) hundreds or thousands of orders into 100 slots of a few items each
- Trick: find a hash function that spreads customers evenly
- Last two digits work, why not first two?

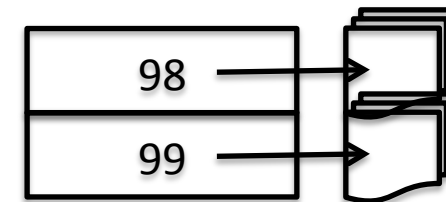
Slots behind desk



•

•

•

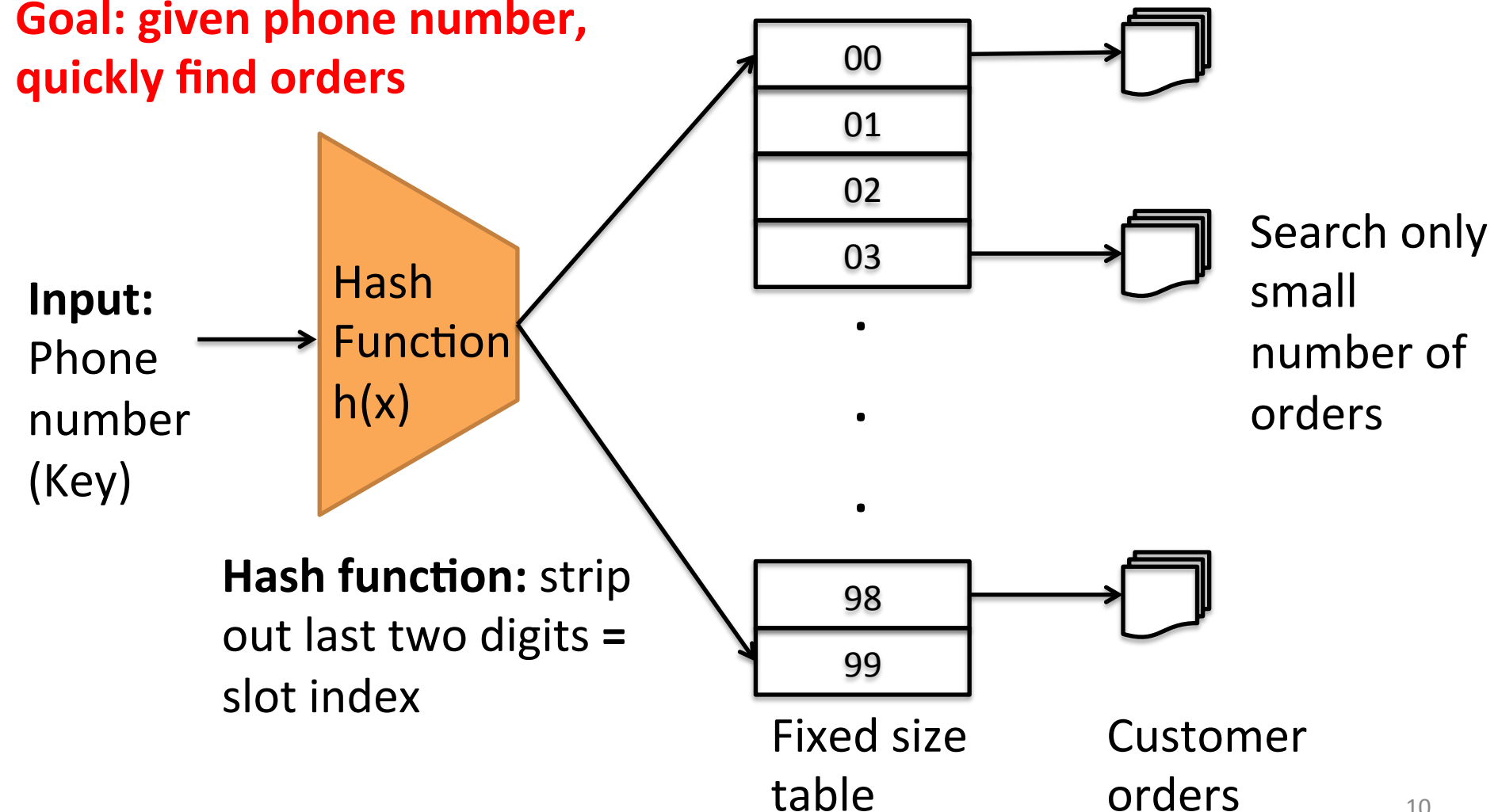


Fixed size  
table

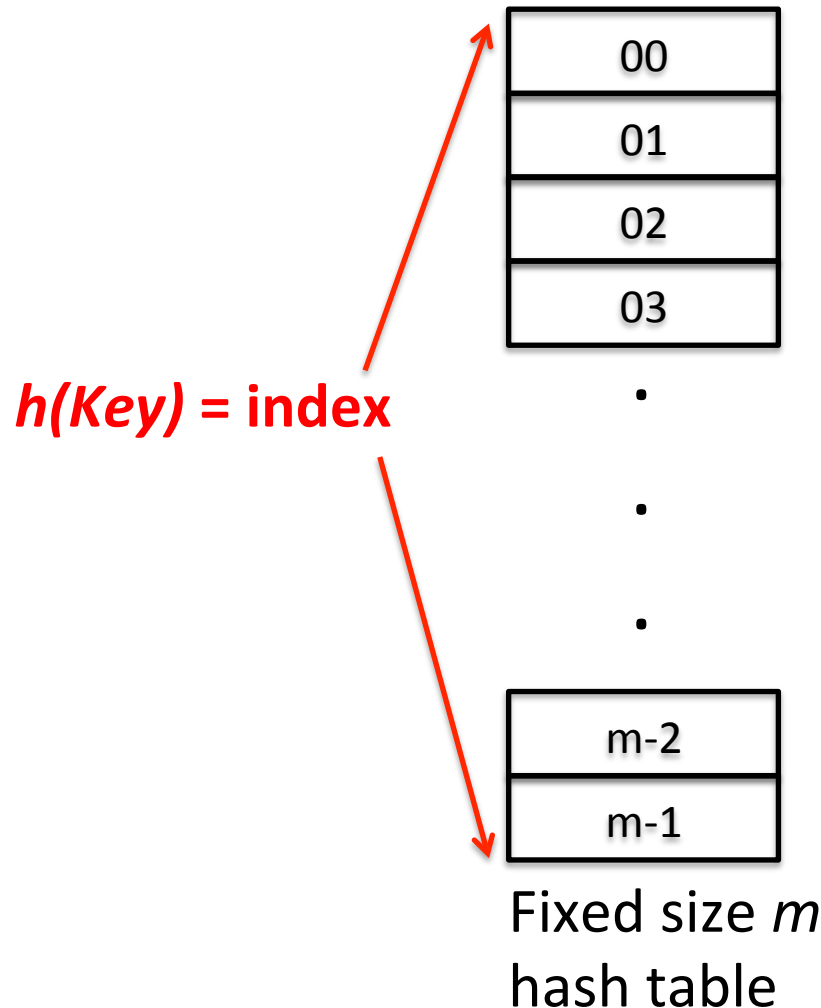
# The store is using a form of hashing based on customer's phone number

Hashing phone numbers to find orders

**Goal: given phone number, quickly find orders**



# Hashing's big idea: map a Key to an array index, then access is fast

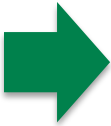


## Map hash table implementation

- Begin with array of fixed size  $m$  (called a hash table)
- Each array index holds item we want to find (e.g., warehouse location of customer's order)
- Use hash function  $h$  on  $\text{Key}$  to give index into hash table
- **$h(\text{Key}) = \text{table index } i = 0..m-1$**
- Get item from hash table at index given by hash function
- Fast to *get/set/add/remove* items
- What about a HashSet?
- Use object itself as Key
- How to hash Key or object?

# Agenda

1. Hashing

- 
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
1. Chaining
  2. Open Addressing

# Good hash functions map keys to indexes in table with three desirable properties

## **Desirable properties of a hash function**

1. Hash can be computed quickly and consistently
2. Hash spreads the universe of keys evenly over the table
3. Small changes in the key (e.g., changing a character in a string or order of letters) should result in different hash value

## **Cryptographic hash function also:**

- Difficult to determine key given the result of hash
- Unlikely that different keys will result in same hash
- We will not focus on crypto requirements

# Suppose we used the first letter of people's names to hash, how would that work?

## First letter of name as hash

1. It can be computed quickly

**Yes**

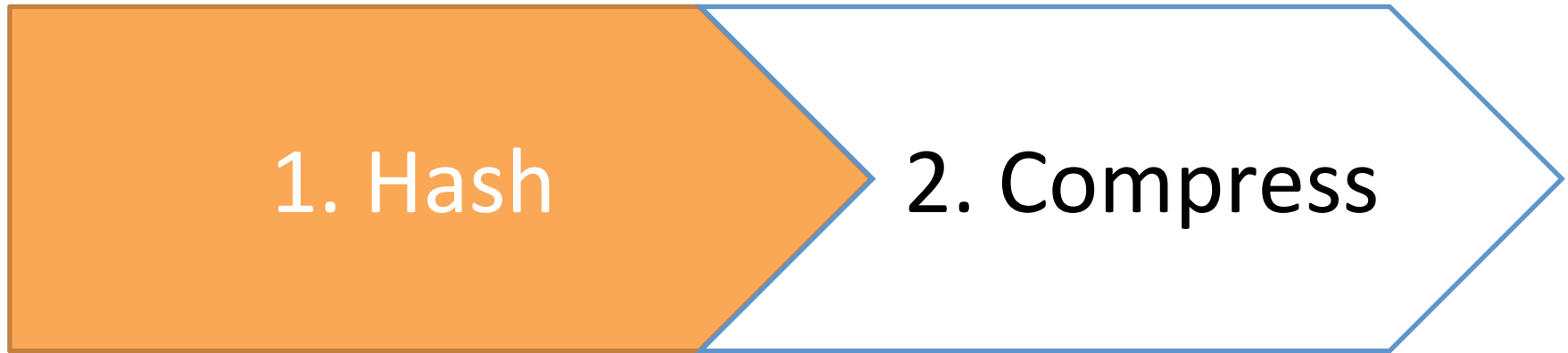
2. It spreads the universe of keys evenly over the table

**No**

3. Small changes in the key (e.g., changing a character in a string or order of letters) should result in different hash value

**Not really. Different, if change first letter, otherwise not.**

# Hashing is often done in two steps: hash then compress



- Get an integer representation of Key
- Integer could be in range  $-\infty$  to  $+\infty$


Constrain integer to table index  $[0..m)$

# First step in hashing is to get an integer representation of the key

**Goal: given key compute an index into hash table array**

**Some Java objects can be directly cast to integers**

- byte
- short
- int
- char



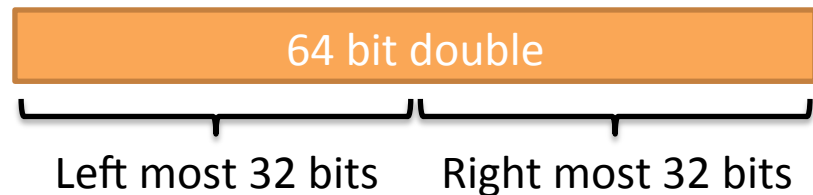
```
char a = 'a';  
int b = (int)a;
```



**b = 97**

**Some items too long cast to integers**

- double (64 bits)
- long (64 bits)
- Too long to make 32 bit integers



**XOR each half**



# Complex objects as Strings can also be hashed to a single integer

## Hashing complex objects

- Consider String  $x$  of length  $n$  where  $x = x_0x_1...x_{n-2}x_{n-1}$
- Pick prime number  $a$  (book recommends 31, 37, or 41)
- Cast each character in  $x$  to an integer
- Calculate polynomial hashCode as  $x_0a^{n-1} + x_1a^{n-2} + ... x_{n-2}a + x_{n-1}$
- Use Horner's rule to efficiently compute hash code

```
public int hashCode() {  
    final int a=37;  
    int sum = x[0]; //first item in array  
    for (int j=1;j<n;j++) {  
        sum = a*sum + x[j]; //array element j  
    }  
    return sum;  
}
```

- Experiments show that when using  $a$  as above, 50,000 English words had fewer than 7 collisions

# Good news: Java provides a *hashCode()* method to compute hashes for us!

## ***hashCode()***

Java does the hashing for us for Strings and autoboxed types with *hashCode()* method

```
Character a = 'a';  
a.hashCode() returns 97
```

```
String b = "Hello";  
b.hashCode() returns 69609650
```

# Bad news: We need to override *hashCode()* and *equals()* for our own Objects

- By default Java uses memory address of objects as a *hashCode*
- But we typically want to hash based on properties of object, not whatever memory location an object happened to be assigned
- This way two objects with same instance variables will hash to the same table location (those objects are considered equal)
- Java says that two *equal* objects must return same *hashCode()*

```
2 public class BlobHash extends Blob{
3
4     @Override
5     public boolean equals(Object otherBlob) {
6         Blob b = (Blob)otherBlob; //cast as Blob
7         if (x == b.x && y == b.y && r == b.r)
8             return true;
9         return false;
10    }
11
12    @Override
13    public int hashCode() {
14        final int a=37;
15        int sum = a*a + (int)x;
16        sum += a * (int)y;
17        sum += (int)r;
18        return sum;
19    }
20 }
```

Here we consider two Blobs *equal* if they have the same *x*, *y* and *r* values  
*equals()* IS THE RIGHT WAY TO COMPARE OBJECT EQUALITY (not ==)

Override *hashCode()* to provide the same hash if two Blobs are *equal*

If don't override *hashCode()* then even though two objects are considered equal, Java will look in the wrong slot

# Java *hashCode()* example

## hashCode()

Some types can be directly cast to an integer

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

Casting 'a' to int is: 97

# Java *hashCode()* example

## hashCode()

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

Some types can be directly cast to an integer

Java does this for us for autoboxed types with *hashCode()*

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

Casting 'a' to int is: 97  
hashCode for 'a' is: 97

# Java *hashCode()* example

## hashCode()

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

Some types can be directly cast to an integer

Java does this for us for autoboxed types with *hashCode()*

*hashCode()* also works for more complex built-in types

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650
```

# Java *hashCode()* example

## hashCode()

```
4 public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

For our own objects, we can provide our own *hashCode()* otherwise we get the memory location by default

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650
```

# Java *hashCode()* example

## hashCode()

For our own objects, we can provide our own *hashCode()* otherwise we get the memory location by default

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

Casting 'a' to int is: 97  
hashCode for 'a' is: 97  
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5.0, 5.0, 5.0  
b2 is at (x,y,r): 0.0, 0.0, 5.0  
hashCode b1: 1564 b2:1374

```
@Override
public int hashCode() {
    final int a=37;
    int sum = a*a + (int)x;
    sum += a * (int)y;
    sum += (int)r;
    return sum;
}
```

*hashCode()* should compute hash:

1. Quickly and consistently
2. Spread keys evenly
3. Small changes = different hash



# Java *equals()* example

## *equals()*

Override *equals()* to test if objects are equivalent  
Otherwise *equals()* checks if same memory location

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

```
@Override
public boolean equals(Object otherBlob) {
    Blob b = (Blob)otherBlob; //cast as Blob
    if (x == b.x && y == b.y && r == b.r)
        return true;
    return false;
}
```

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650
```

```
b1 is at (x,y,r): 5.0, 5.0, 5.0
b2 is at (x,y,r): 0.0, 0.0, 5.0
hashCode b1: 1564 b2:1374
```

# Java *equals()* example

## `equals()`

Override *equals()* to test if objects are equivalent  
Otherwise *equals()* checks if same memory location

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

```
@Override
public boolean equals(Object otherBlob) {
    Blob b = (Blob)otherBlob; //cast as Blob
    if (x == b.x && y == b.y && r == b.r)
        return true;
    return false;
}
```

This is the right way to  
compare if two  
objects are equivalent  
(not `b1 == b2`)

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

Casting 'a' to int is: 97  
hashCode for 'a' is: 97  
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5.0, 5.0, 5.0  
b2 is at (x,y,r): 0.0, 0.0, 5.0  
hashCode b1: 1564 b2:1374  
b1 is equal to b2: false

# Java *equals()* example

## `equals()`

Override *equals()* to test if objects are equivalent  
Otherwise *equals()* checks if same memory location

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

```
@Override
public boolean equals(Object otherBlob) {
    Blob b = (Blob)otherBlob; //cast as Blob
    if (x == b.x && y == b.y && r == b.r)
        return true;
    return false;
}
```

This is the right way to  
compare if two  
objects are equivalent  
(not `b1 == b2`)

After updating x,y, and r  
two Blobs are now equal

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650
```

```
b1 is at (x,y,r): 5.0, 5.0, 5.0
b2 is at (x,y,r): 0.0, 0.0, 5.0
hashCode b1: 1564 b2:1374
b1 is equal to b2: false
after update b1 now equals b2: true
```

# Java *equals()* example

## `equals()`

Override *equals()* to test if objects are equivalent  
Otherwise *equals()* checks if same memory location

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

```
@Override
public boolean equals(Object otherBlob) {
    Blob b = (Blob)otherBlob; //cast as Blob
    if (x == b.x && y == b.y && r == b.r)
        return true;
    return false;
}
```

This is the right way to  
compare if two  
objects are equivalent  
(not `b1 == b2`)

After updating x,y, and r  
two Blobs are now equal

*hashCode()* also returns the same  
value for equivalent objects

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy  
<terminated> HashTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_112.jdk/Contents/Home/bin/java (Jan 28, 2018, 5:52:17 PM)

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5.0, 5.0, 5.0
b2 is at (x,y,r): 0.0, 0.0, 5.0
hashCode b1: 1564 b2:1374
b1 is equal to b2: false
after update b1 now equals b2: true
hashCode b1: 1564 b2:1564
```

# Java *equals()* example

## *equals()*

Override *equals()* to test if objects are equivalent  
Otherwise *equals()* checks if same memory location

```
4= public static void main(String[] args) {
5     char a = 'a';
6     int b = (int)a;
7     System.out.println("Casting 'a' to int is: " + b);
8     Character z = 'a';
9     System.out.println("hashCode for 'a' is: " + z.hashCode());
10    String y = "Hello";
11    System.out.println("hashCode for 'hello' is: " + y.hashCode());
12    System.out.println();
13
14    //create new Blob with overridden equals and hashCode functions
15    BlobHash b1 = new BlobHash();
16    b1.x = 5; b1.y = 5; b1.r = 5; //update b1's location
17    BlobHash b2 = new BlobHash(); //create new HashBlob
18    System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
19    System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
20    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
21    System.out.println("b1 is equal to b2: " + b1.equals(b2));
22    b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
23    System.out.println("after update b1 now equals b2: " + b1.equals(b2));
24    System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
25 }
26 }
```

```
@Override
public boolean equals(Object otherBlob) {
    Blob b = (Blob)otherBlob; //cast as Blob
    if (x == b.x && y == b.y && r == b.r)
        return true;
    return false;
}
```

This is the right way to  
compare if two  
objects are equivalent  
(not `b1 == b2`)

After updating x,y, and r  
two Blobs are now equal

HashMap and HashSet will  
now put equivalent objects  
in the same slot in the  
table (after compression)

hashCode() also returns the same  
value for equivalent objects

# Hashing is often done in two steps: hash then compress

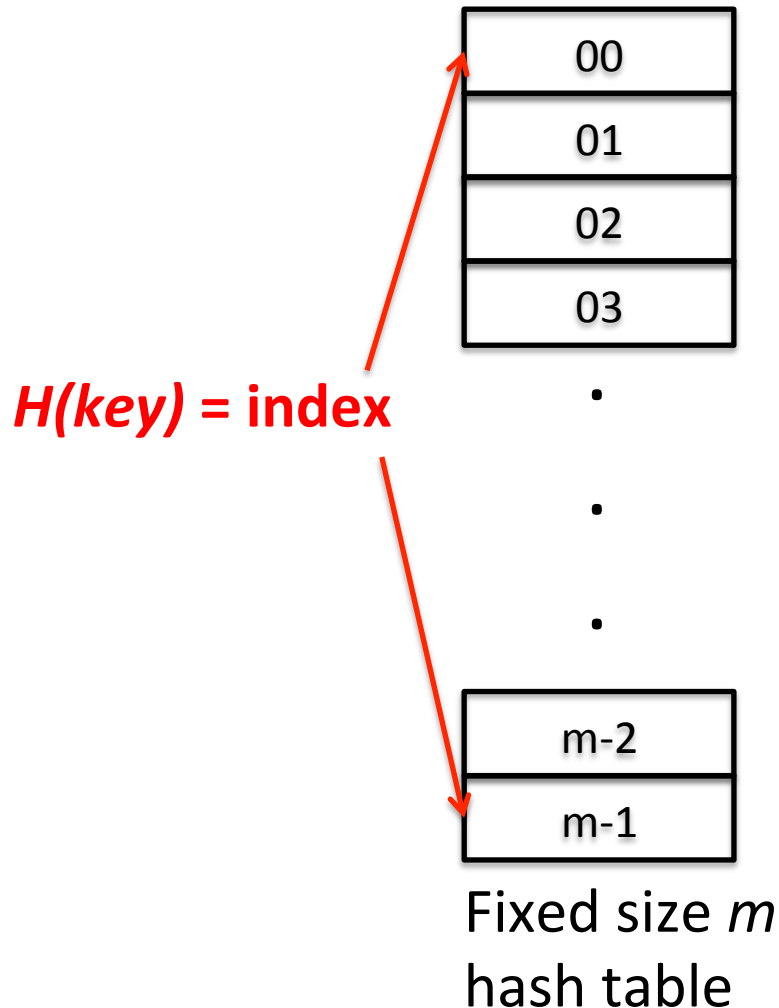


- Get an integer representation of Key
- Integer could be in range  $-\infty$  to  $+\infty$

Constrain integer to table index  $[0..m)$




# May have to compress hash value to table index [0..m)



## Compressing

- $hashCode()$  value may be larger than the table (or negative!)
- Need to constrain value to one of the table slots  $[0..m)$
- “Division method” is simple:  
$$h(key) = key \% m$$
- Works well if  $m$  is prime
- Book gives a more advanced version called Multiply-Add-And-Divide (MAD)
- Java takes care of this for us 😊
- Eventually will encounter collisions where multiple keys map to the same slot ☹️

# Agenda

1. Hashing
2. Computing Hash functions
-  3. Implementing Maps/Sets with hashing
4. Handling collisions
  1. Chaining
  2. Open Addressing



# Map methods can be easily implemented with hashing

## *put(key, value)*

- Hash key to get table index
  - Get  $i = \text{key.hashCode}()$
  - Compress  $i$  to  $0..m-1$
- Store value

## *get(key)*

- Hash key to get table index
  - Get  $i = \text{key.hashCode}()$
  - Compress  $i$  to  $0..m-1$
- Return stored value

## *remove(key)*

- Hash key to get table index
  - Get  $i = \text{key.hashCode}()$
  - Compress  $i$  to  $0..m-1$
- Remove stored value

### Open questions:


- What if multiple items hash to the same index?
- What if table fills up?



0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
-  4. Handling collisions
  1. Chaining
  2. Open Addressing

# Collisions happen when multiple keys map to the same table index

## Integer keys

Given table size  $m = 13$

`put(key,value)`

- Hash & constrain key
- Store value at index

Compute  **$h(\text{key}) = (\text{key} \% m)$**

0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Collisions happen when multiple keys map to the same table index

## Integer keys

Given table size  $m = 13$

`put(key,value)`

- Hash & constrain key
- Store value at index

Compute  $h(\text{key}) = (\text{key} \% m)$

Example

- $h(6) = 6$

0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Collisions happen when multiple keys map to the same table index

## Integer keys

Given table size  $m = 13$

`put(key,value)`

- Hash & constrain key
- Store value at index

Compute  $h(\text{key}) = (\text{key} \% m)$

Example

- $h(6) = 6$
- **$h(8) = 8$**

0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Collisions happen when multiple keys map to the same table index

## Integer keys

Given table size  $m = 13$

`put(key,value)`

- Hash & constrain key
- Store value at index

Compute  $h(\text{key}) = (\text{key} \% m)$

Example

- $h(6) = 6$
- $h(8) = 8$
- **$h(16) = 3$**

0
1
2
16
4
5
6
7
8
9
10
11
12

$m = 13$

# Collisions happen when multiple keys map to the same table index

## Integer keys

Given table size  $m = 13$

`put(key,value)`

- Hash & constrain key
- Store value at index

Compute  $h(\text{key}) = (\text{key} \% m)$

Example

- $h(6) = 6$
- $h(8) = 8$
- $h(16) = 3$
- **$h(19) = 6$**

0
1
2
16
4
5
6
7
8
9
10
11
12

Collision!  
6 and 19 mapped to  
the same index

$$h(6) = h(19)$$

$m = 13$

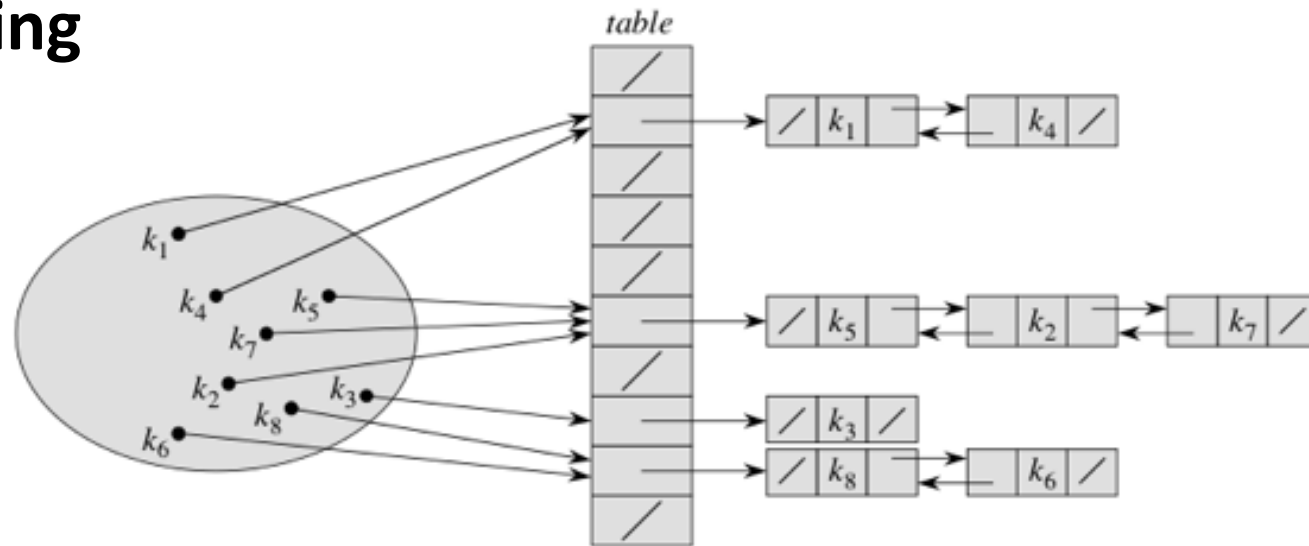
# Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
  - ➡ 1. Chaining
  - 2. Open Addressing



# Chaining handles collisions by creating a linked list for each table entry

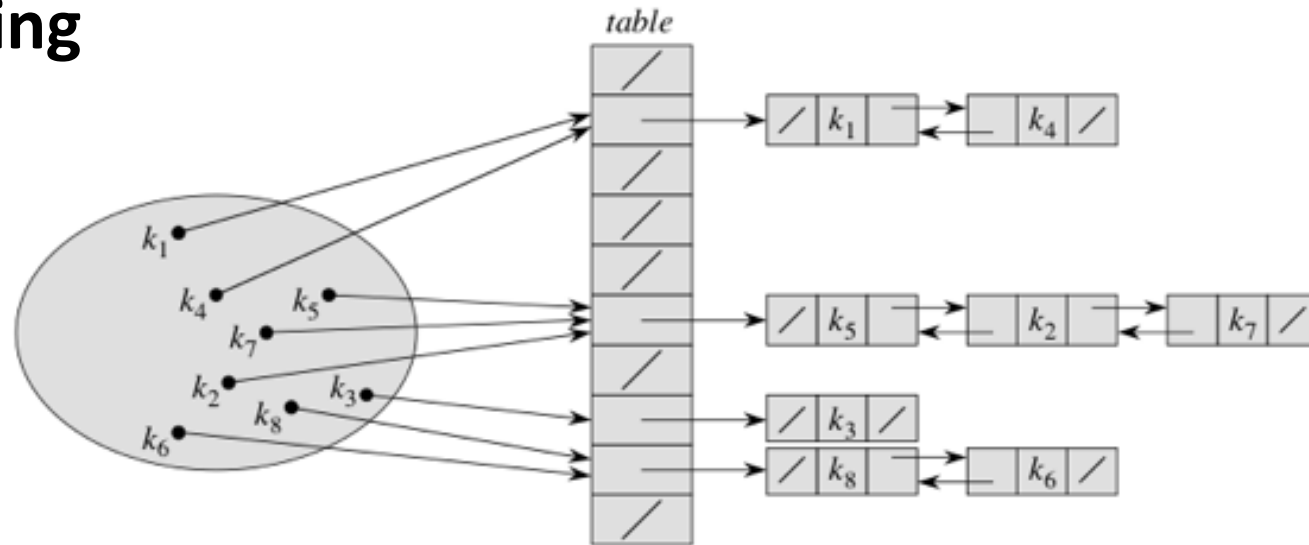
## Chaining



- Create a table pointing to linked list of items that hash to the same index (similar to last class word positions)
- Slot  $i$  holds all keys  $k$  for which  $h(k) = i$
- Splice in new elements at head for  $O(1)$  performance
- NOTE: Values associated with Keys are not shown, here just showing Keys

# Load factor measures number of items in the list that must be searched on average

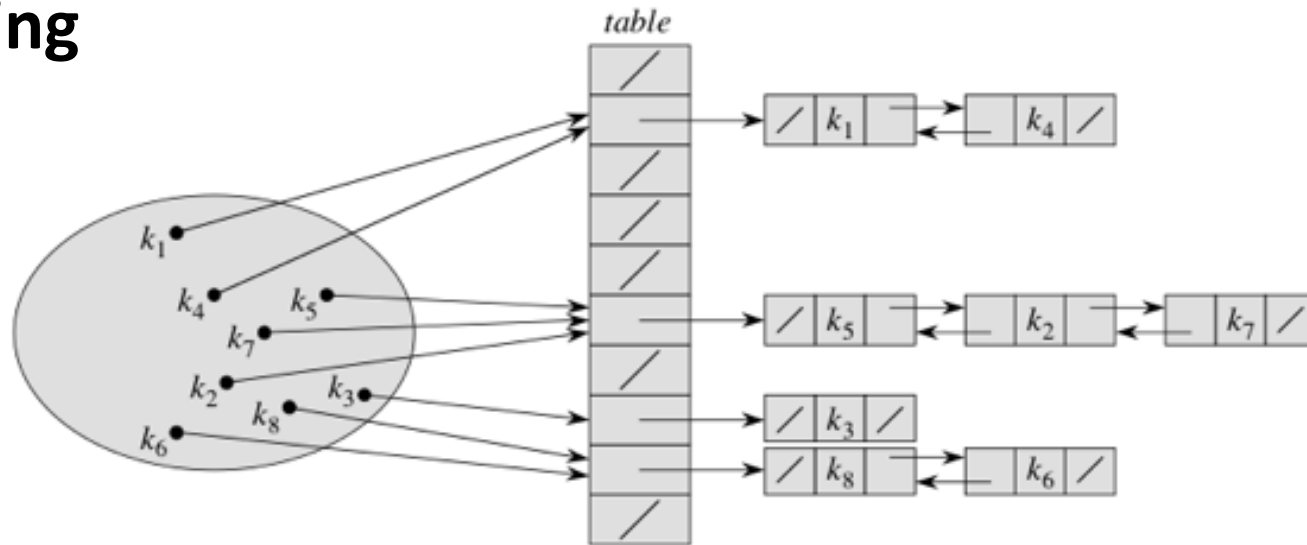
## Chaining



- Assume table with  $m$  slots and  $n$  keys are stored in it
- On average, we expect  $n/m$  elements per collision list
- This is called the **load factor** ( $\lambda = n/m$ )
- Expected search time is  $\Theta(1 + \lambda)$ , assuming **simple uniform hashing** (each possible key equally likely to hash into a particular slot), worst case  $\Theta(n)$  if bad hash function


# If the load factor gets too high, then we should increase the table size

## Chaining



- If  $n$  (# elements) becomes larger than  $m$  (table size), then collisions are inevitable and search time goes up
- Java increases table size by 2X and *rehashes* into new table when  $\lambda > 0.75$  to combat this problem
- Problem: memory fragmentation with link lists spread out all over, might not be good for embedded systems

# Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
  1. Chaining
  -  2. Open Addressing

# Open addressing is different solution, everything is stored in the table itself

## Open addressing using linear probing

- Insert item at hashed index (no linked list)
- For key  $k$  compute  $h(k)=i$ , insert at index  $i$
- If collision, a simple solution is called ***linear probing***
  - Try inserting at  $i+1$
  - If slot  $i+1$  full, try  $i+2...$  until find empty slot
  - Wrap around to slot 0 if hit end of table at  $m-1$
  - If  $\lambda < 1$  will find empty slot
  - If  $\lambda \approx 1$ , increase table size ( $m*2$ ) and rehash
- Search analogous to insertion, compute key and probe until find item or empty slot (key not in table)

# Linear probing is one way of handling collisions under open addressing

## Integer keys

Given table size  $m = 13$

Compute  $h(\text{key}) = (\text{key} \% m)$

### Example

- $h(6) = 6$
- $h(8) = 8$
- $h(16) = 3$

0
1
2
16
4
5
6
7
8
9
10
11
12

$m = 13$

# Linear probing is one method of open addressing

## Integer keys

Given table size  $m = 13$

Compute  $h(\text{key}) = (\text{key} \% m)$

### Example

- $h(6) = 6$
- $h(8) = 8$
- $h(16) = 3$
- **$h(19) = 6$**

0
1
2
16
4
5
6
7
8
9
10
11
12

Collision!

$m = 13$

# Linear probing is one method of open addressing

## Integer keys

Given table size  $m = 13$

Compute  $h(\text{key}) = (\text{key} \% m)$

### Example

- $h(6) = 6$
- $h(8) = 8$
- $h(16) = 3$
- $h(19) = 6$

0
1
2
16
4
5
6
19
8
9
10
11
12

$m = 13$

Insert at  $i+1 = 7$

To find items later,  
hash to table index,  
then probe until find  
item or hit empty  
slot



# Deleting items is tricky, need to mark deleted spot as available but not empty

## Problems deleting items under linear probing

- Insert  $k_1$ ,  $k_2$ , and  $k_3$  where  $h(k_1)=h(k_2)=h(k_3)$
- All three keys hash to the same slot in this example
- $k_1$  in slot  $i$ ,  $k_2$  in slot  $i+1$ ,  $k_3$  in slot  $i+2$
- Remove  $k_2$ , creates hole at  $i+1$
- Search for  $k_3$ 
  - Hash  $k_3$  to  $i$ , slot  $i$  holds  $k_1 \neq k_3$ , advance to slot  $i+1$
  - Find hole at  $i+1$ , assume  $k_3$  not in hash table
- Can mark deleted spaces as available for insertion, and search skips over marked spaces
- This can be a problem if many deletes create many marked slots, search approaches linear time

# Clustering of keys can build up and reduce performance

## Clustering problem

- Long runs of occupied slots (clusters) can build up increasing search and insert time
- Clusters happen because empty slot preceded by  $t$  full slots gets filled with probability  $(t+1)/m$ , instead of  $1/m$  (e.g.,  $t$  keys can now fill open slot instead of just 1 key)
- Clusters can bump into each other exacerbating the problem

# Clustering of keys can build up and reduce performance

## Integer keys

Given table size  $m = 13$

Compute  $h(\text{key}) = (\text{key} \% m)$

### Example

- $h(6) = 6$
- $h(8) = 8$
- $h(16) = 3$
- $h(19) = 6$

0
1
2
16
4
5
6
19
8
9
10
11
12

Hashing 6,7,8, or 9 go into index 9

Makes index 9 more likely to be filled than other slots

$m = 13$

# Double hashing can help with the clustering problem

## Double hashing

- **Big idea: instead of stepping by 1 at each collision like linear probing, step by a different amount where the step size depends on the key**
- Use two hash functions  $h_1$  and  $h_2$  to make a third  $h'$
- $h'(k,p) = (h_1(k) + ph_2(k)) \bmod m$ , where  $p$  number of probes
- First probe  $h_1(k)$ ,  $p=0$ , then  $p$  incremented by 1 on each collision until space is found
- Result is a step by  $h_2(k)$  on each collision (then mod  $m$  to stay inside table size), instead of 1
- Need to design hashes so that if  $h_1(k_1) = h_1(k_2)$ , then ***unlikely***  $h_2(k_1) = h_2(k_2)$

# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k,p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	p	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$

0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k,p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	p	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$
8	0	8	9	$(8+0*9)\%13 = 8$

0
1
2
3
4
5
6
7
8
9
10
11
12

$m = 13$

# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k, p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	p	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$
8	0	8	9	$(8+0*9)\%13 = 8$
16	0	3	5	$(3+0*5)\%13 = 3$

0
1
2
16
4
5
6
7
8
9
10
11
12

$m = 13$

# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k, p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	p	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$
8	0	8	9	$(8+0*9)\%13 = 8$
16	0	3	5	$(3+0*5)\%13 = 3$
19	0	6	8	$(6+0*8)\%13 = 6$

0
1
2
16
4
5
6
7
8
9
10
11
12

**Collision!**

$m = 13$



# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

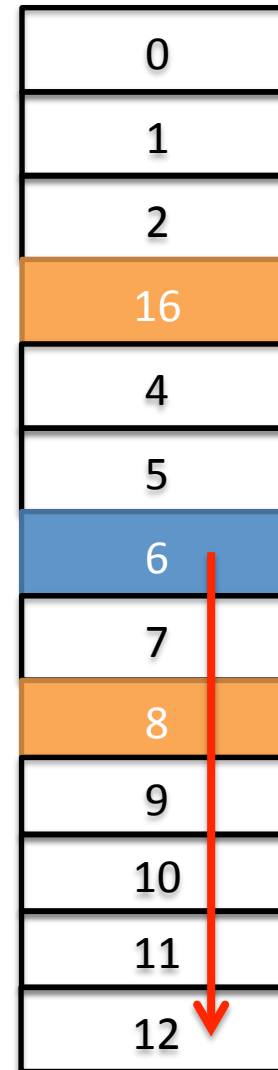
$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k, p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	$p$	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$
8	0	8	9	$(8+0*9)\%13 = 8$
16	0	3	5	$(3+0*5)\%13 = 3$
19	0	6	8	$(6+0*8)\%13 = 6$
19	<b>1</b>	6	<b>8</b>	$(6+\mathbf{1*8})\%13 = \mathbf{1}$



**Collision!**

**Increment  $p$**

**Step forward  
by  $h_2(\text{key}) = 8$   
spaces**

**Wrap around  
if needed** <sup>57</sup>

# Double hashing can help with the clustering problem

## Integer keys

Given table size  $m = 13$

Compute

$$h_1(\text{key}) = (\text{key} \% m)$$

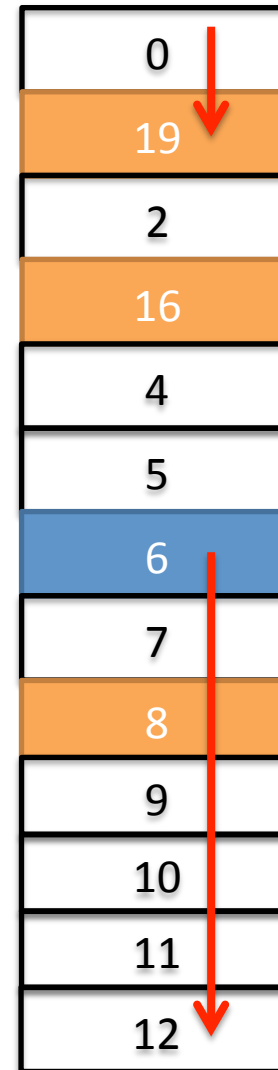
$$h_2(\text{key}) = 1 + (\text{key} \% (m-1))$$

$$h'(k, p) = (h_1(k) + ph_2(k)) \% m$$

$h_1$  same as before  
 $h_2$  new hash function  
 $p$  = probe number  
(initially 0)

## Example

Key	$p$	$h_1$	$h_2$	$h'$
6	0	6	7	$(6+0*7)\%13 = 6$
8	0	8	9	$(8+0*9)\%13 = 8$
15	0	2	4	$(2+0*4)\%13 = 2$
19	0	6	8	$(6+0*8)\%13 = 6$
19	<b>1</b>	6	<b>8</b>	$(6+\mathbf{1*8})\%13 = \mathbf{1}$



Insert here

Collision!

Increment  $p$

Step forward  
by  $h_2(\text{key}) = 8$   
spaces

Wrap around  
if needed <sup>58</sup>

# Run time degrades as $\lambda$ gets large, so keep $\lambda$ small by growing hash table

## Expected insert and search time

- Average number of probes is approximately  $1/(1-\lambda)$
- As  $\lambda \rightarrow 1$ , expected number of probes becomes large, when  $\lambda$  small, number of probes approaches 1
- If table 90% full, then expect about 10 probes for unsuccessful search
- Successful search generally a little faster, about 2.5 probes (math on course web page and in book)
- Must grow table and rehash when copying to new table to keep the table sparsely populated or performance suffers

# Assuming load factor $\lambda$ is small and hashing spreads keys, core operations are $O(1)$

Operation	<u>Expected</u> run time	Notes
<i>hash(k)</i>	$O(1)$	<ul style="list-style-type: none"><li>• Math operations on key <math>k</math> to hash and compress, outputs <math>0 \dots m-1</math></li><li>• Constant time, does not depend on number of items in Set or Map</li></ul>

# Assuming load factor $\lambda$ is small and hashing spreads keys, core operations are $O(1)$

Operation	<u>Expected</u> run time	Notes
<i>hash(k)</i>	$O(1)$	<ul style="list-style-type: none"><li>• Math operations on key <math>k</math> to hash and compress, outputs <math>0 \dots m-1</math></li><li>• Constant time, does not depend on number of items in Set or Map</li></ul>
<i>find(k)</i>	$O(1)$	<ul style="list-style-type: none"><li>• Once have index of table due to hash:<ul style="list-style-type: none"><li>• Chaining: traverse linked list <math>O(\lambda) = O(1)</math></li><li>• Probing: probe until find <math>O(1/(1-\lambda)) = O(1)</math></li></ul></li></ul>

# Assuming load factor $\lambda$ is small and hashing spreads keys, core operations are $O(1)$

Operation	Expected run time	Notes
$hash(k)$	$O(1)$	<ul style="list-style-type: none"><li>Math operations on key <math>k</math> to hash and compress, outputs <math>0 \dots m-1</math></li><li>Constant time, does not depend on number of items in Set or Map</li></ul>
$find(k)$	$O(1)$	<ul style="list-style-type: none"><li>Once have index of table due to hash:<ul style="list-style-type: none"><li>Chaining: traverse linked list <math>O(\lambda) = O(1)</math></li><li>Probing: probe until find <math>O(1/(1-\lambda)) = O(1)</math></li></ul></li></ul>
$get(k)$	$O(1+1) = O(1)$	<ul style="list-style-type: none"><li><i>Hash + find</i>:</li><li>chaining = <math>O(1+\lambda) = O(1)</math>, probing = <math>O(1+(1/(1-\lambda))) = O(1)</math></li></ul>

# Assuming load factor $\lambda$ is small and hashing spreads keys, core operations are $O(1)$

Operation	Expected run time	Notes
$hash(k)$	$O(1)$	<ul style="list-style-type: none"><li>Math operations on key <math>k</math> to hash and compress, outputs <math>0 \dots m-1</math></li><li>Constant time, does not depend on number of items in Set or Map</li></ul>
$find(k)$	$O(1)$	<ul style="list-style-type: none"><li>Once have index of table due to hash:<ul style="list-style-type: none"><li>Chaining: traverse linked list <math>O(\lambda) = O(1)</math></li><li>Probing: probe until find <math>O(1/(1-\lambda)) = O(1)</math></li></ul></li></ul>
$get(k)$	$O(1+1) = O(1)$	<ul style="list-style-type: none"><li><i>Hash + find</i>:</li><li>chaining = <math>O(1+\lambda) = O(1)</math>, probing = <math>O(1+(1/(1-\lambda))) = O(1)</math></li></ul>
$put(k,v)$	$O(1)$ <u><math>+O(1)</math></u> $O(1)$	<ul style="list-style-type: none"><li><i>Hash + find</i> = <math>O(1)</math></li><li>Plus update or add element:<ul style="list-style-type: none"><li>Chaining: update value or add at head <math>O(1)</math></li><li>Probing: store value in array <math>O(1)</math></li></ul></li></ul>

# Assuming load factor $\lambda$ is small and hashing spreads keys, core operations are $O(1)$

Operation	Expected run time	Notes
$hash(k)$	$O(1)$	<ul style="list-style-type: none"> <li>Math operations on key <math>k</math> to hash and compress, outputs <math>0 \dots m-1</math></li> <li>Constant time, does not depend on number of items in Set or Map</li> </ul>
$find(k)$	$O(1)$	<ul style="list-style-type: none"> <li>Once have index of table due to hash: <ul style="list-style-type: none"> <li>Chaining: traverse linked list <math>O(\lambda) = O(1)</math></li> <li>Probing: probe until find <math>O(1/(1-\lambda)) = O(1)</math></li> </ul> </li> </ul>
$get(k)$	$O(1+1) = O(1)$	<ul style="list-style-type: none"> <li><i>Hash + find</i>:</li> <li>chaining = <math>O(1+\lambda) = O(1)</math>, probing = <math>O(1+(1/(1-\lambda))) = O(1)</math></li> </ul>
$put(k,v)$	$O(1)$ $+O(1)$ $O(1)$	<ul style="list-style-type: none"> <li><i>Hash + find</i> = <math>O(1)</math></li> <li>Plus update or add element: <ul style="list-style-type: none"> <li>Chaining: update value or add at head <math>O(1)</math></li> <li>Probing: store value in array <math>O(1)</math></li> </ul> </li> </ul>
$remove(k)$	$O(1)$ $+O(1)$ $O(1)$	<ul style="list-style-type: none"> <li><i>Hash + find</i> = <math>O(1)</math></li> <li>Plus remove element: <ul style="list-style-type: none"> <li>Chaining: update one pointer <math>O(1)</math></li> <li>Probing: mark space empty <math>O(1)</math></li> </ul> </li> </ul>

**Assuming a small load factor and uniform hashing, the core operations of HashSets and HashMaps are constant time!**



