

# CS151 Intro to Data Structures

## Hashmaps

# Announcements & Outline

No lab today. HW6 and Lab8 due Sunday

Only two homeworks left after this one!

CS Major Info session - today at 4pm Park 231

Today:

- Map review
- Hash functions and some cool applications
- Hash Maps - a magical data structure

# Maps

- Also called “dictionaries” or “associative arrays”
- Similar syntax to an array:
  - `m[key]` retrieves a value
  - `m[key] = value` assigns a value
  - keys need not be ints
- data structure that stores a collection of key-value pairs

# Map ADT

- `get(k)` : if the map  $M$  has an entry with key  $k$ , return its associated value; else, return null
- `put(k, v)` : insert entry  $(k, v)$  into the map  $M$ ; if key  $k$  is not already in  $M$ , then return null; else, replace old value with  $v$  and return old value associated with  $k$
- `remove(k)` : if the map  $M$  has an entry with key  $k$ , remove it from  $M$  and return its associated value; else, return null
- `size()`, `isEmpty()`
- `keySet()` : return an iterable collection of the keys in  $M$
- `values()` : return an iterator of the values in  $M$
- `entrySet()` : return an iterable collection of the entries in  $M$

# Map.Entry Interface

- A (Key, Value) pair
- Keys and Values can be any reference type
- Methods:
  - `getKey()`
  - `getValue()`
  - `setValue(V val)`
- **Implementation:** `SimpleEntry`

# ArrayMap

Last class we implemented a `Map` as an array of `SimpleEntry` s

Implementation & Runtime complexity?

- `put`
- `get`
- `remove`

# HashMaps

# Hash Functions

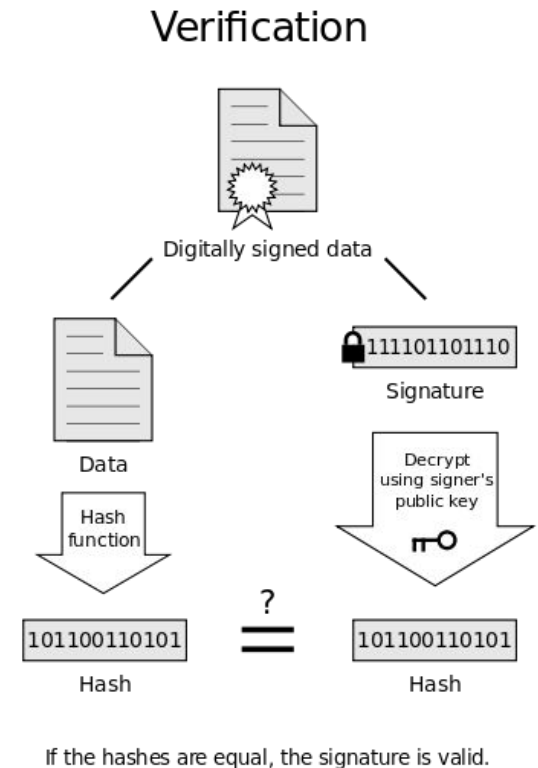
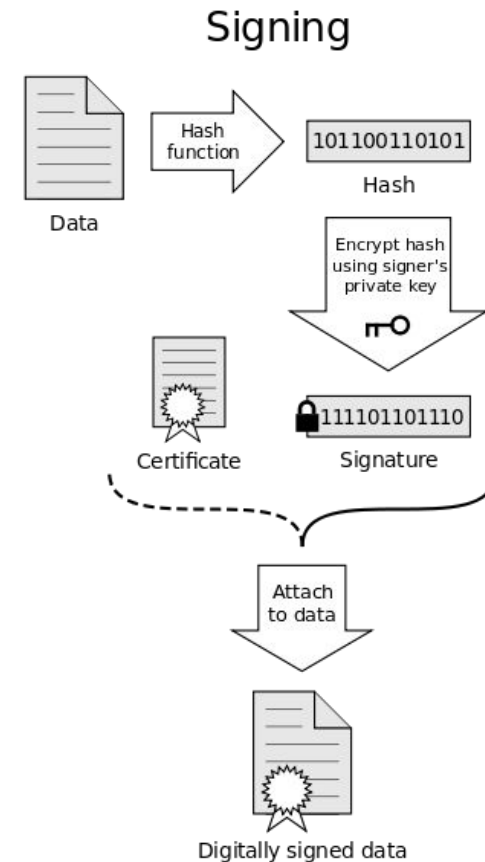
- A *hash function* maps an arbitrary length input to a fixed length *unique* output
- <https://emn178.github.io/online-tools/sha256.html>
- Applications
  - data structures
  - encryption / digital signatures
  - blockchain
- Properties of a good hash function:
  - one way
  - **collision resistant**
  - uniformity
  - quick to compute



# Digital Signatures

cryptographic method used to validate the *authenticity* and *integrity* of a digital message or document

- In **public key cryptography** Each user has two keys:
  - one **public key** (e, n)
  - one **private key** (d, n)
- RSA encryption:
  - Using the sender's private key,
  - encrypt the **hash** of the message or document M.
  - **Signature = Hash(M) ^ d mod n**
- RSA Decryption
  - Using the sender's public key,
  - decrypt the message
  - **hash = Signature ^ e mod n**



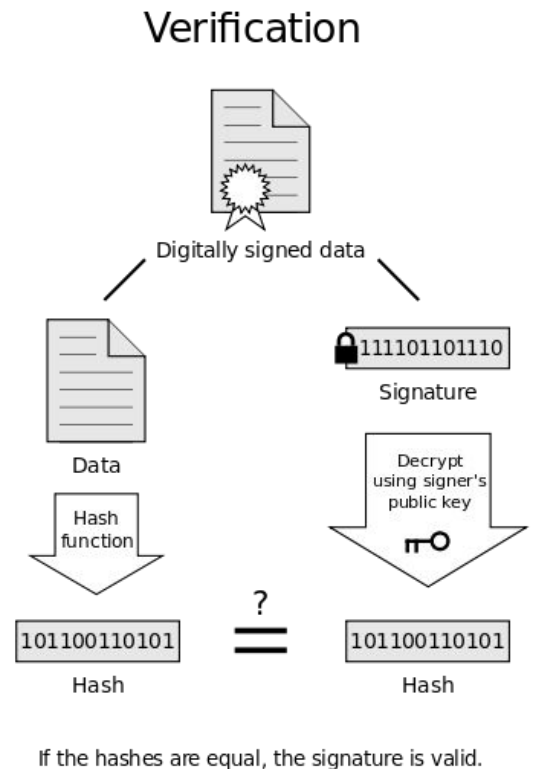
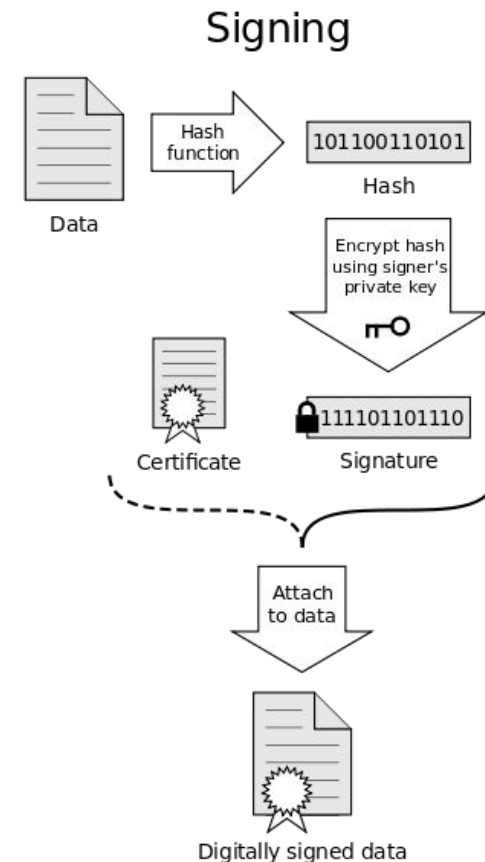
# Digital Signatures

cryptographic method used to validate the *authenticity* and *integrity* of a digital message or document

Why is hashing necessary?

1. Fixed length output
  - a. Encryption scheme handles a small predictably sized amount of data
  - b. More efficient to sign the hash than the entire document / message
2. Data Integrity
  - a. Any small change in the message will result in a completely different hash value

Question: why do I need to send the original data **and** the signature? Why can't I recover the document from the signature?



# Hash Functions on the Blockchain

1. Digital signatures on each transaction
2. Data integrity and immutability
  - a. decentralized computing - every machine needs to agree on the state of the program
  - b.  $O(n)$  algorithm to verify every memory slot is the same
  - c. Hash is easier to check against

Back to data structures :)

# A Simple Hash Function

Given an int  $x$ ...

$h(x)$  = last 4 digits of  $x$

- one way?
- collision resistant?
- uniform?

# Another Simple Hash Function

$$h(x) = x \% N$$

- one way?
- collision resistant?
- uniform?

# HashMaps

- How can we use hash functions to improve the performance of our ArrayMap implementation?
- **Use hash as our array index!**
- How would the following operations look?
  - put
  - get
  - remove

# More on Hash Functions

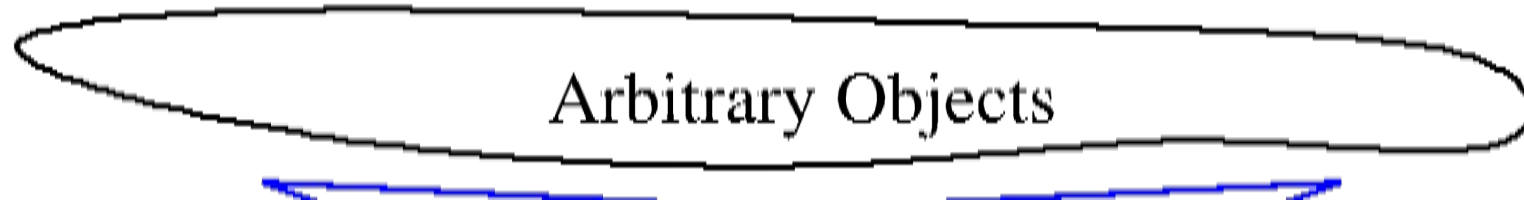
A hash function is usually specified as the composition of two functions:

- hash code:  $h_1: \text{key} \rightarrow \text{integers}$
- compression:  $h_2: \text{integers} \rightarrow [0, N - 1]$
- $h(x) = h_2(h_1(x))$

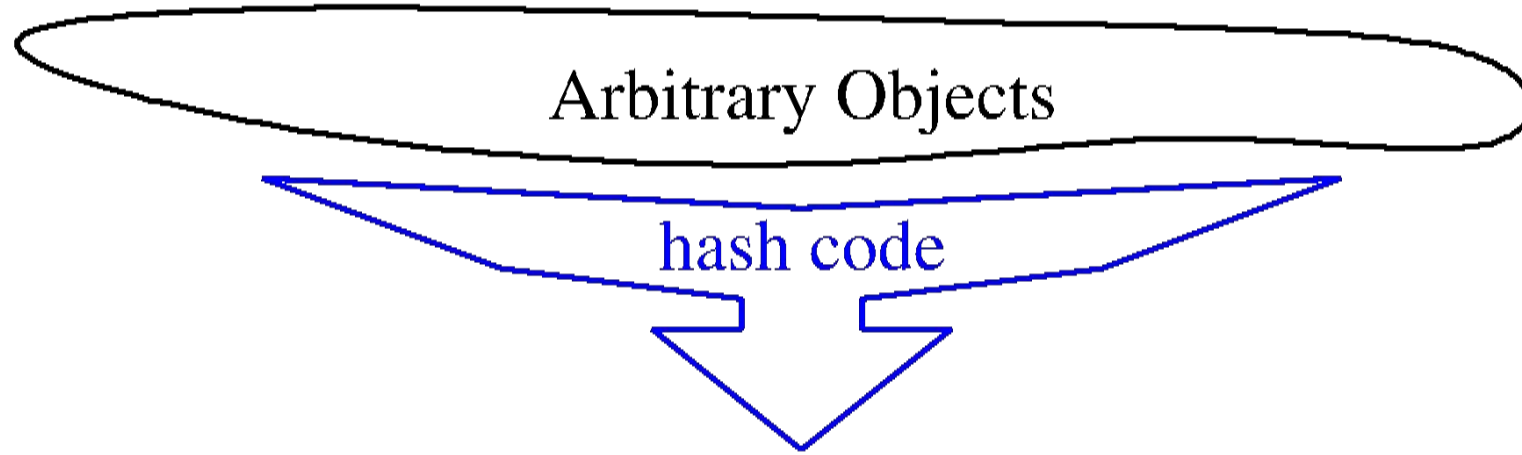
The goal is to "disperse" the keys in an appropriately random way



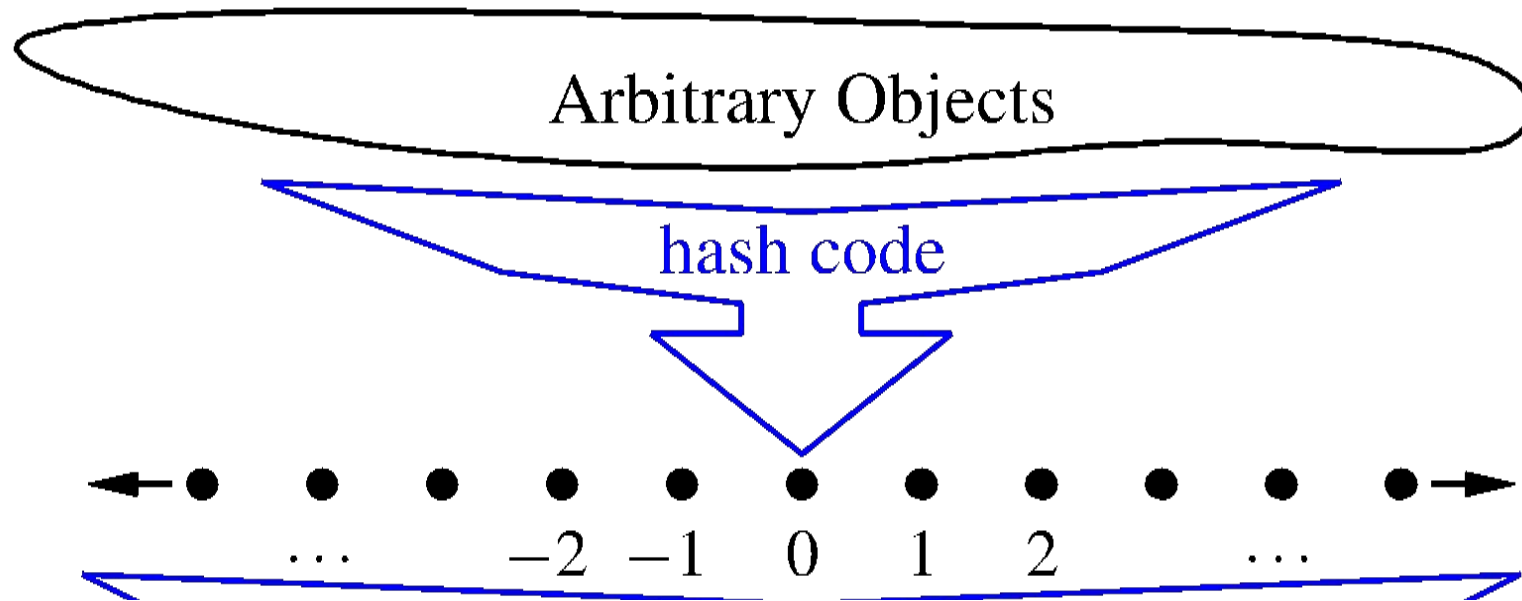
# Hash Function Illustration



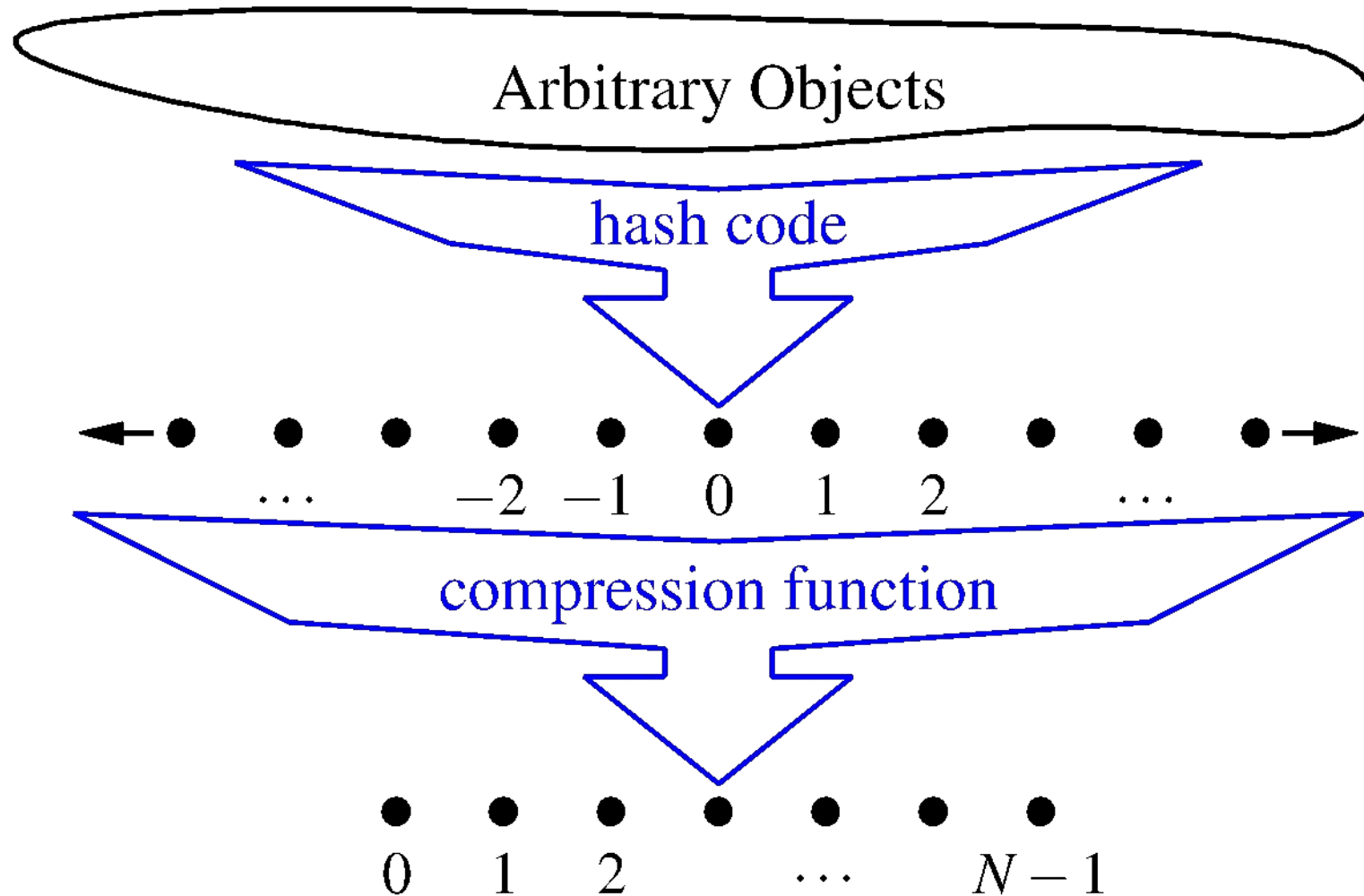
# Hash Function Illustration



# Hash Function Illustration



# Hash Function Illustration



# Hash Codes ( $h_1$ )

- $h_1(k)$  takes an arbitrary key and computes an integer
  - Goal: collision resistant!
  - Need not be a fixed length or in fixed range  $[0, N)$
  - Can even be negative

# Hash Codes $h_1$

## Idea 1: Memory Addresses

- use the memory address where the keys are stored
- **default hash code for Java objects**

<a href="https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html">docs.oracle.com/javase/8/docs/api/java/lang/Object.html</a>	
<b>Method Summary</b>	
<b>All Methods</b>	<b>Instance Methods</b>
<b>Concrete Methods</b>	
<b>Modifier and Type</b>	<b>Method and Description</b>
protected <b>Object</b>	<b>clone()</b> Creates and returns a copy of this object.
boolean	<b>equals(Object obj)</b> Indicates whether some other object is "equal to" this on
protected void	<b>finalize()</b> Called by the garbage collector on an object when garba object.
<b>Class&lt;?&gt;</b>	<b>getClass()</b> Returns the runtime class of this Object.
int	<b>hashCode()</b> Returns a hash code value for the object.

# Hash Codes ( $h_1$ )

What if our key is not an object?

- Integer cast: `byte`, `short`, `int`, `char` and `float`
- What about `long` and `double`??
  - Can't cast to `int`. We'll lose information!
  - COLLISIONS
  - Instead, partition bits into `int` components and combine them

# Compression ( $h_2$ )

Why do we need compression?

## Compression Idea 1: mod

$$h_2(x) = x \bmod N$$

forces output to be in range  $[0, N)$

How should we choose  $N$ ?

primes!



# Compression ( $h_2$ )

Compression Idea 2: Multiply Add and Divide (MAD)

$$h_2(x) = ((ax + b) \bmod p) \bmod N$$

where  $N$  is the capacity

$p$  is a prime  $> N$

$a$  and  $b$  are  $[0, p)$

$a$  scales the range

$b$  shifts the start

# HashMap

Book's `AbstractHashMap` hash method uses:

$$h_1(k) = k.\text{hashCode}() \text{ // java memory address}$$
$$h_2(x) = ((ax + b) \% p) \% N$$

# Hash Maps

Efficient data structure that stores (Key, Value) pairs

Implements the Map ADT

- `get (k)` : if the map `M` has an entry with key `k`, return its associated value; else, return null
- `put (k, v)` : insert entry `(k, v)` into the map `M`; if key `k` is not already in `M`, then return null; else, replace old value with `v` and return old value associated with `k`
- `remove (k)` : if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null
- `size ()`, `isEmpty ()`
- `keySet ()` : return an iterable collection of the keys in `M`
- `values ()` : return an iterator of the values in `M`
- `entrySet ()` : return an iterable collection of the entries in `M`

# Hash Maps

## Implementation

Let's start with our ArrayMap and use hashes for indices

**BE CAREFUL!**    %    MEANS REMAINDER IN JAVA NOT MOD!

What should we do if there's a collision?

- For a first impl, let's just overwrite

# Performance Analysis

	ArrayMap	Collision Resistant Hash Map
get		
put		
remove		

# AbstractHashMap

```
1 public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2     protected int n = 0;           // number of entries in the dictionary
3     protected int capacity;        // length of the table
4     private int prime;              // prime factor
5     private long scale, shift;      // the shift and scaling factors
6     public AbstractHashMap(int cap, int p) {
7         prime = p;
8         capacity = cap;
9         Random rand = new Random();
10        scale = rand.nextInt(prime-1) + 1;
11        shift = rand.nextInt(prime);
12        createTable();
13    }
14    public AbstractHashMap(int cap) { this(cap, 109345121); } // default prime
15    public AbstractHashMap() { this(17); } // default capacity
16    // public methods
17    public int size() { return n; }
18    public V get(K key) { return bucketGet(hashValue(key), key); }
19    public V remove(K key) { return bucketRemove(hashValue(key), key); }
20    public V put(K key, V value) {
21        V answer = bucketPut(hashValue(key), key, value);
22        if (n > capacity / 2) // keep load factor <= 0.5
23            resize(2 * capacity - 1); // (or find a nearby prime)
24        return answer;
25    }
26 }
```

# AbstractHashMap

```
26 // private utilities
27 private int hashCode(K key) {
28     return (int) ((Math.abs(key.hashCode())*scale + shift) % prime) % capacity);
29 }
30 private void resize(int newCap) {
31     ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32     for (Entry<K,V> e : entrySet())
33         buffer.add(e);
34     capacity = newCap;
35     createTable(); // based on updated capacity
36     n = 0; // will be recomputed while reinserting entries
37     for (Entry<K,V> e : buffer)
38         put(e.getKey(), e.getValue());
39 }
40 // protected abstract methods to be implemented by subclasses
41 protected abstract void createTable();
42 protected abstract V bucketGet(int h, K k);
43 protected abstract V bucketPut(int h, K k, V v);
44 protected abstract V bucketRemove(int h, K k);
45 }
```

# Handling Collisions



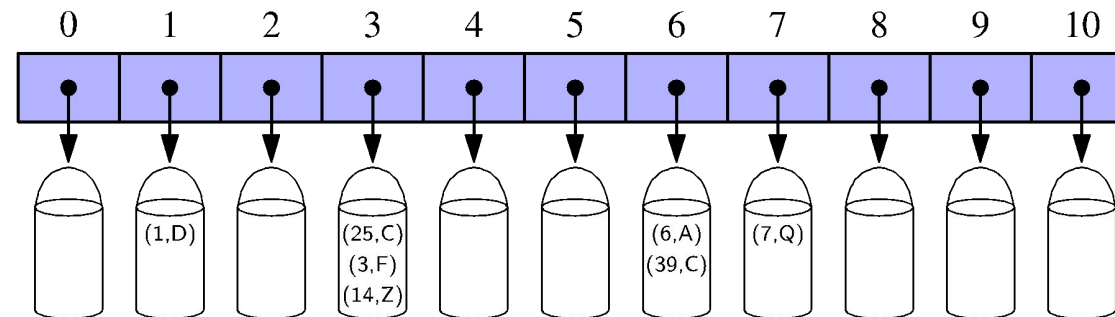
# Handling Collisions

A hash function does not guarantee one-to-one mapping – no hash function does

One approach **chaining**:

When more than one key hash to the same index, we have a bucket

Each index holds a collection of entries



# Collision Handling

Collisions occur when elements with different keys are mapped to the same cell

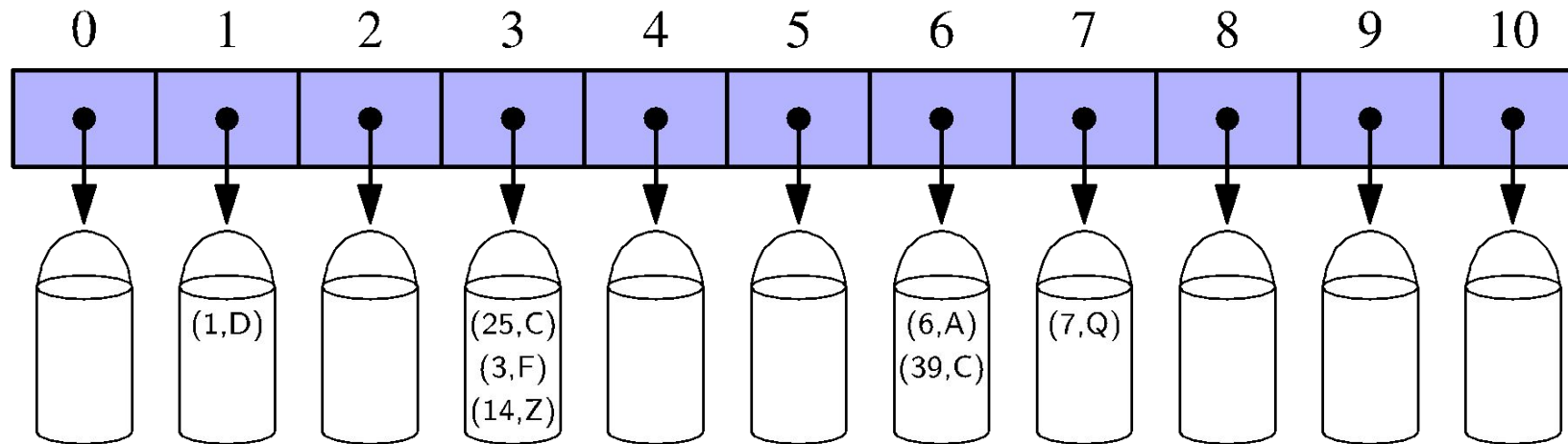
**Separate Chaining:** let each cell in the table point to a linked list of entries that map there

Simple, but requires additional memory besides the table

# Let's implement a ChainHashMap

What data structure should we use for the buckets?

- LinkedList!



# Summary

## Hash Map:

- Efficient data structure with constant time\* access, insertion, and removal
- \* assuming no collisions or expansions

## Hash Functions:

- Composition of  $h_1$  and  $h_2$
- $h_2$  compresses output of  $h_1$  between 0 and  $N$

# Collision Handling Approach #2

## **Open Addressing and Probing**

When a collision occurs, find an empty slot nearby to store the colliding element

# Open Addressing and Probing

- Example:  $h(x) = x \% 13$
- insert 18(5), 41(2), 22(9), 44(5), 59(7), 32(6), 31(5), 73(8)

Keep “probing”

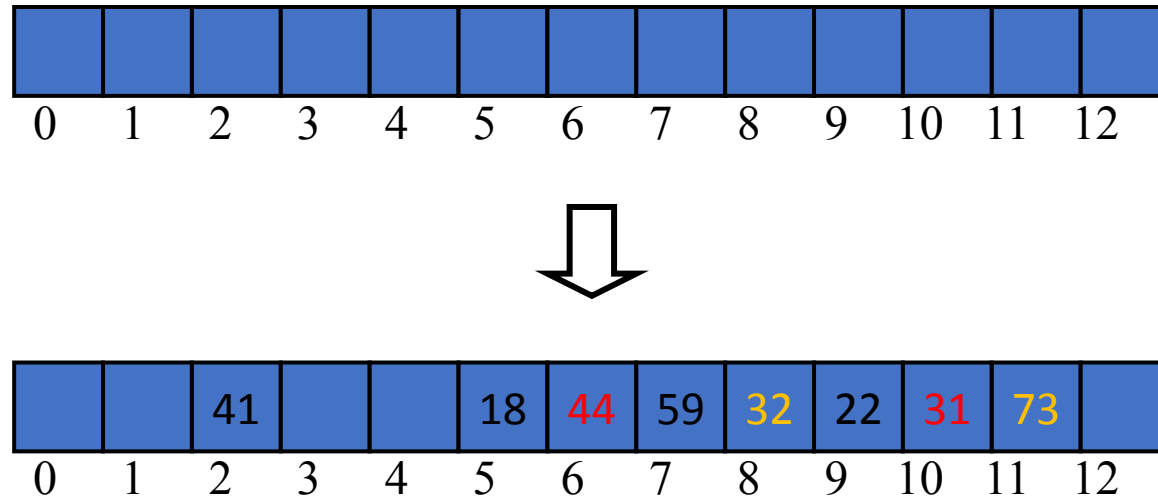
$(h(k)+1)\%n$

$(h(k)+2)\%n$

....

$(h(k)+i)\%n$

until you find an  
empty slot!



# ProbeHashMap

Let's implement a ProbeHashMap

# Open Addressing and Probing

**Linear Probing** (what we just implemented):

- Keep “*probing*” until you find an empty slot
$$(h(k)+1) \% n$$
$$(h(k)+2) \% n$$
$$\dots$$
$$(h(k)+i) \% n$$
- Colliding items cluster together – future collisions to cause a longer sequence of probes



# Open Addressing and Probing

## Quadratic Probing:

- Keep “*probing*” until you find an empty slot

$$(h(k) + f(1)) \% n$$

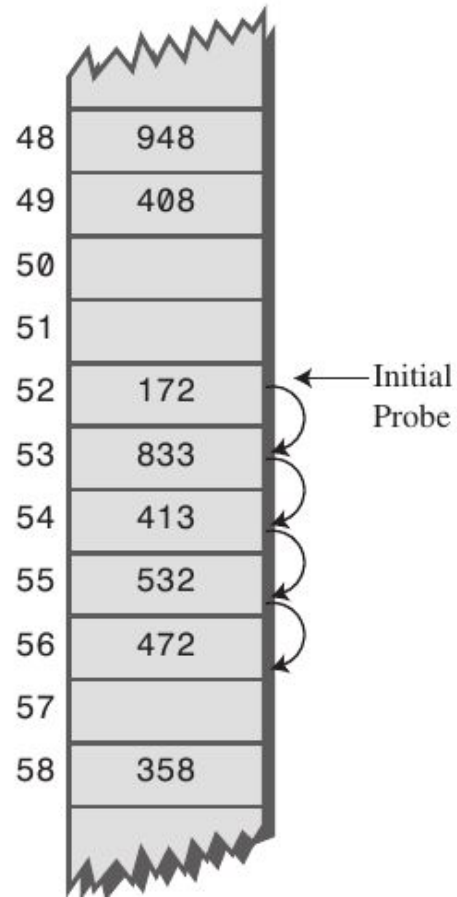
$$(h(k) + f(2)) \% n$$

....

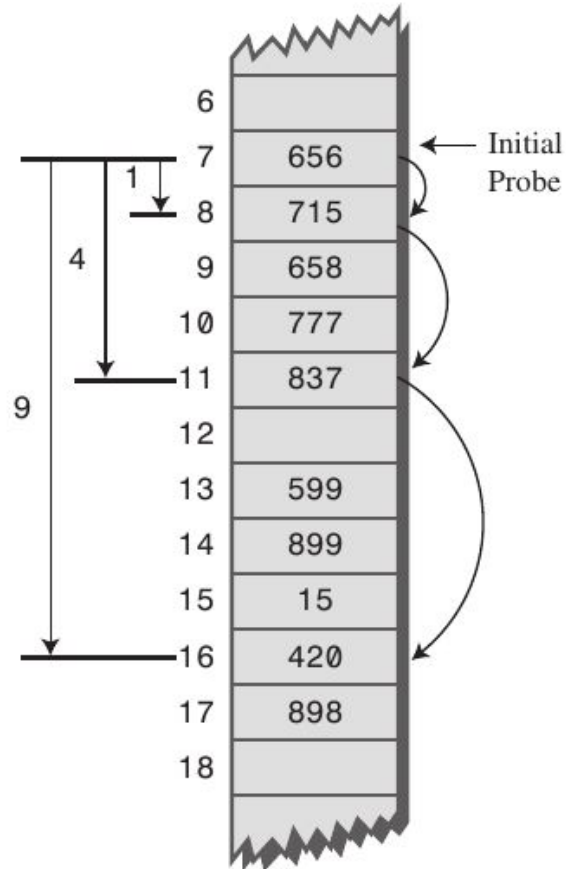
$$(h(k) + f(i)) \% n$$

where  $f(i) = i^2$

# Linear Probing vs Quadratic Probing



Linear Probing



Quadratic Probing

- Quadratic probing still creates large clusters!
- Unlike linear probing, they are clustered away from the initial hash position
- If the primary hash index is  $x$ , probes go to  $x+1$ ,  $x+4$ ,  $x+9$ ,  $x+16$ ,  $x+25$  and so on, this results in **Secondary Clustering**

# Approach #3: Double Hashing

Let's try to avoid clustering.

To probe, let's use a **second hash function**

- Keep “*probing*” until you find an empty slot

$$(h(k) + f(1)) \% n$$

$$(h(k) + f(2)) \% n$$

....

$$(h(k) + f(i)) \% n$$

Where  $f(i) = i * h'(k)$

# Approach #3: Double Hashing

Keep “*probing*” until you find an empty slot

$$(h(k) + f(1)) \% n$$

$$(h(k) + f(2)) \% n$$

....

$$(h(k) + f(i)) \% n$$

Where  $f(i) = i * h'(k)$

A common choice for  $h'(k) = q - (k \% q)$   
where  $q$  is prime and  $< n$

# Example

$k$	$h(k)$	$h'(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	
31	5	4	5	9 0
73	8	4	8	

- Insert 18, 41, 22, 44, 59, 32, 31, 73

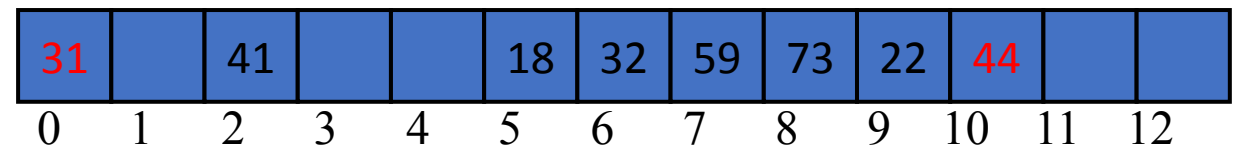
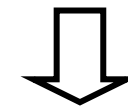
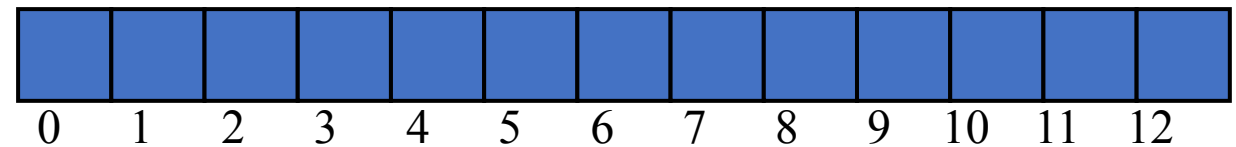
**probe:**

$$(h(k) + f(k)) \% n$$

$$h(k) = k \% 13$$

$$f(k) = i * h'(k)$$

$$h'(k) = 7 - k \% 7$$



# Performance Analysis

	ChainHashMap Best Case	ChainHashMap Worst Case	ProbeHashMap Best Case	ProbeHashMap Worst Case
get				
put				
remove				

Which is better in practice?

# Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list

# Performance Analysis

	ArrayMap	HashMap with good hashing and good probing
get		
put		
remove		



# Performance of Hashtable

	array	linked list	BST (balanced)	HashTable
search				
insert				
remove				

# Load Factor

- HashMaps have an underlying array... what if it gets full?
  - For ChainHashMap collisions increase
  - For ProbeHashMap we need to resize!
- Load Factor = # of elements stored / capacity
- A common strategy is to resize the hash map when the load factor exceeds a predefined threshold (often 0.75)
  - tradeoff between memory and runtime