CS151 Intro to Data Structures

Hashmaps

Announcements & Outline

No lab today. HW6 and Lab8 due Friday

Only two homeworks left after this one!

CS Major Info session - tomorrow at 4pm Park 230

OH on zoom this week - 12-1:30pm

Today:

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

Warm up: Choosing the Right Data Structure…

You are building a task management system that needs to handle various operations efficiently. Each task has an associated importance level, and tasks can be completed, added, or queried frequently. The operations that your system needs to support are:

1. **Add a task**: Insert a new task with a specified importance level.
2. **Remove most important task**: Complete and remove the task with the highest importance.
3. **Get most important task**: View the task with the highest importance without removing it.
4. **Check if tasks are available**: Determine if there are any tasks left to complete.
5. **Update the importance of an existing task**: Change the importance level of a specific task.

Warm up: Choosing the Right Data Structure...

Priority Queue

1. **Add a task**:
    a. O(logn)
2. **Remove most important task**:
    a. O(logn)
3. **Get most important task**:
    a. O(1)
4. **Check if tasks are available**:
    a. O(1)
5. **Update the importance of an existing task**:
    a. O(logn) assuming you have the location of the given task

Warm up: Choosing the Right Data Structure...

You are building a task management system that needs to handle various operations efficiently. Each task has an associated importance level, and tasks can be completed, added, or queried frequently. The operations that your system needs to support are:

1. **Add a task**: Insert a new task with a specified importance level.
2. **Remove most important task**: Complete and remove the task with the highest importance.
3. **Get most important task**: View the task with the highest importance without removing it.
4. **Check if tasks are available**: Determine if there are any tasks left to complete.
5. **Update the importance of an existing task**: Change the importance level of a specific task.

Warm up: Choosing the Right Data Structure...

Priority Queue

1. **Add a task**:
   a. O(logn)
2. **Remove most important task**:
   a. O(logn)
3. **Get most important task**:
   a. O(1)
4. **Check if tasks are available**:
   a. O(1)
5. **Update the importance of an existing task**:
   a. O(logn) assuming you have the location of the given task

Warm up: Choosing the Right Data Structure...

You are building a task management system that needs to handle various operations efficiently. Each task has an associated importance level, and tasks can be completed, added, or queried frequently. The operations that your system needs to support are:

1. **Add a task**: Insert a new task with a specified importance level.
2. **Remove most important task**: Complete and remove the task with the highest importance.
3. **Get most important task**: View the task with the highest importance without removing it.
4. **Check if tasks are available**: Determine if there are any tasks left to complete.
5. **Update the importance of an existing task**: Change the importance level of a specific task.

Warm up: Choosing the Right Data Structure...

Priority Queue

1. **Add a task**:
   a. O(logn)
2. **Remove most important task**:
   a. O(logn)
3. **Get most important task**:
   a. O(1)
4. **Check if tasks are available**:
   a. O(1)
5. **Update the importance of an existing task**:
   a. O(logn) assuming you have the location of the given task

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

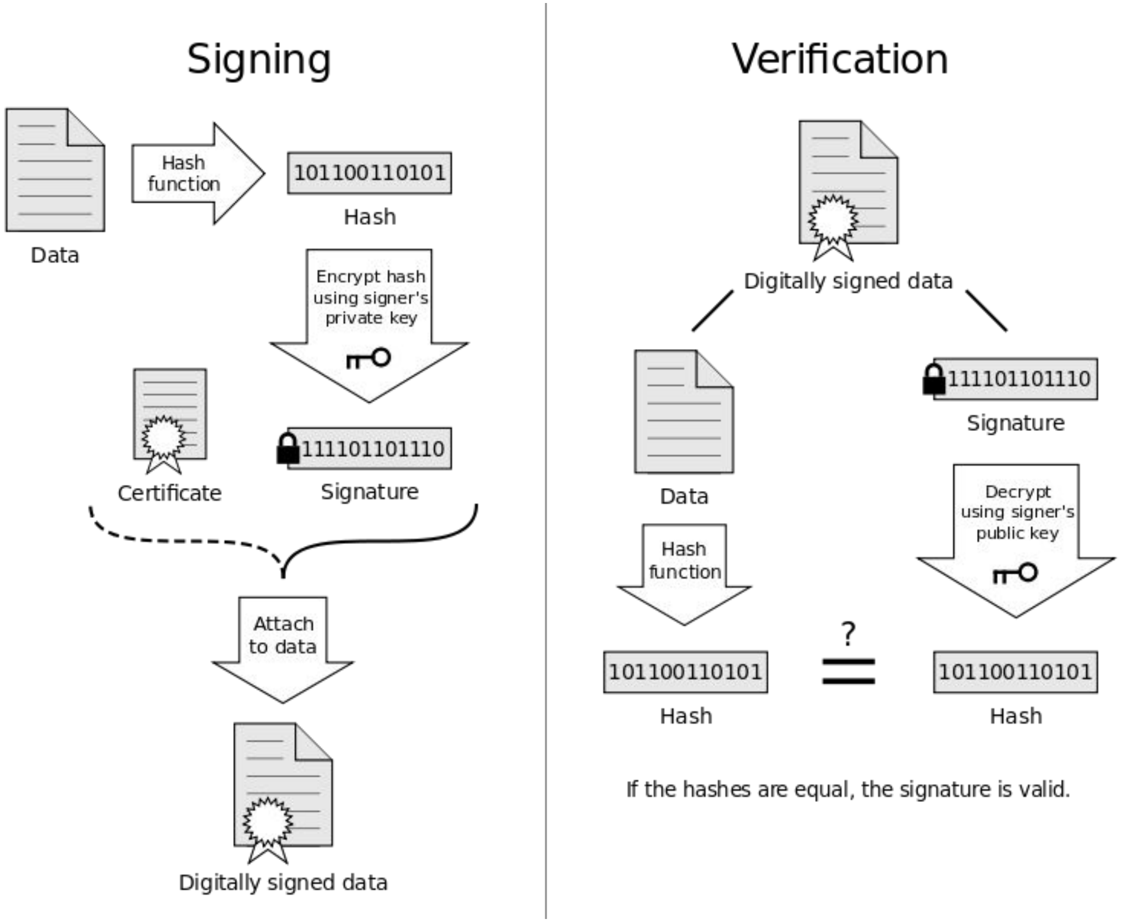
Digital Signatures

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

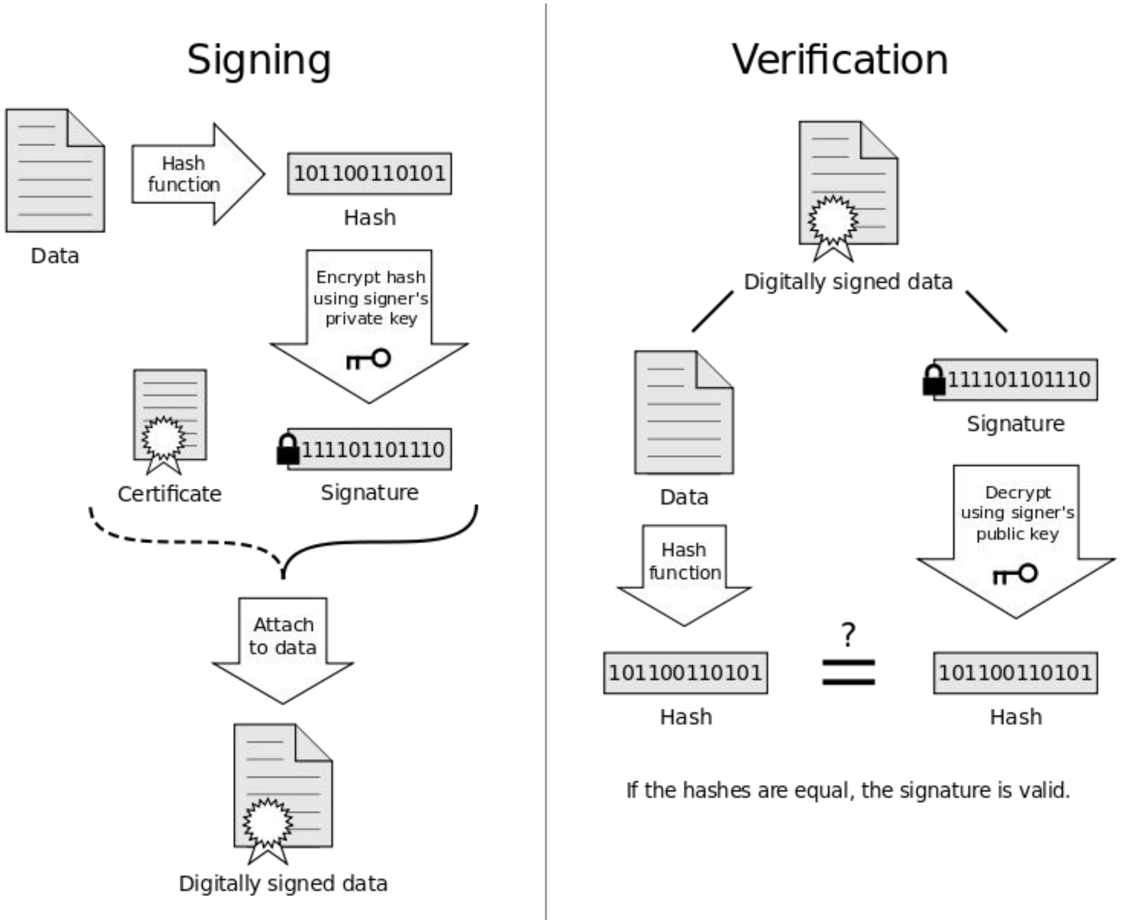
cryptographic method used to validate the *authenticity* and *integrity* of a digital message or document

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

1. Data Integrity
   b. 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

1. 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

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

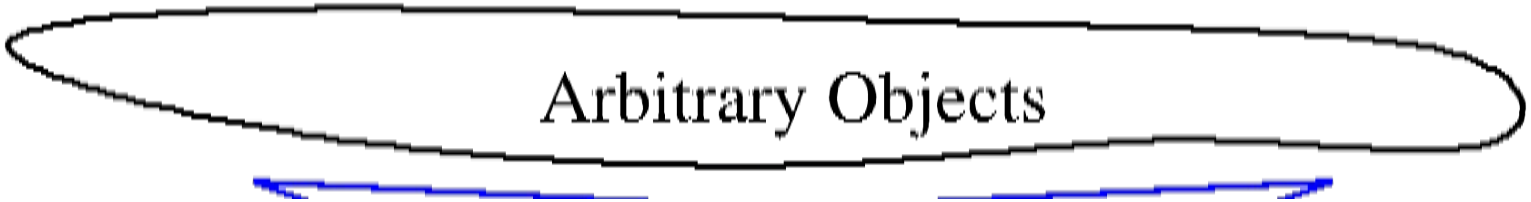
More on Hash Functions

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

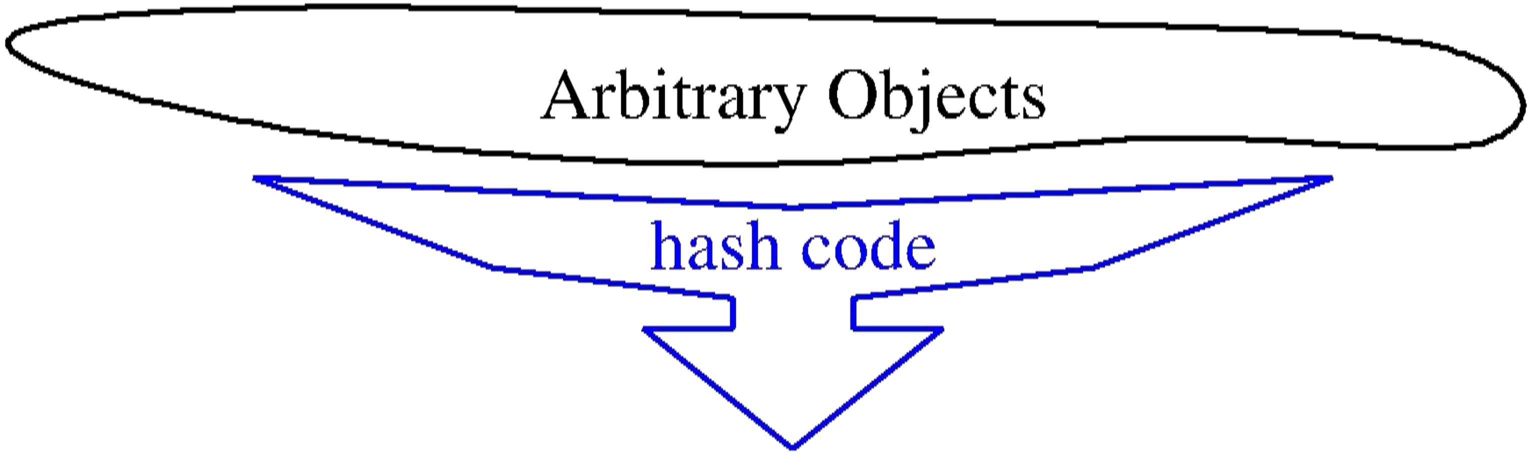
- hash code: h_1: key → integers
- compression: h_2: integers → $[0, N-1]$
- $h(x) = h_2(h_1(x))$

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

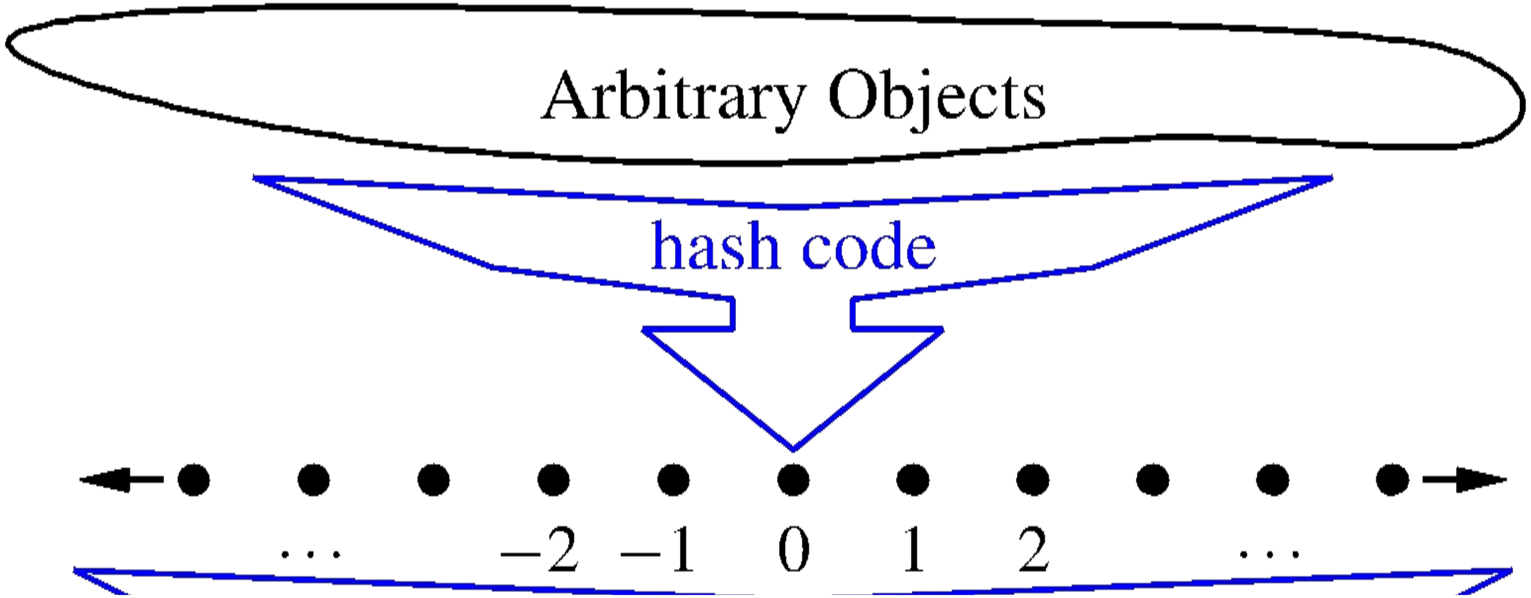
Hash Function Illustration



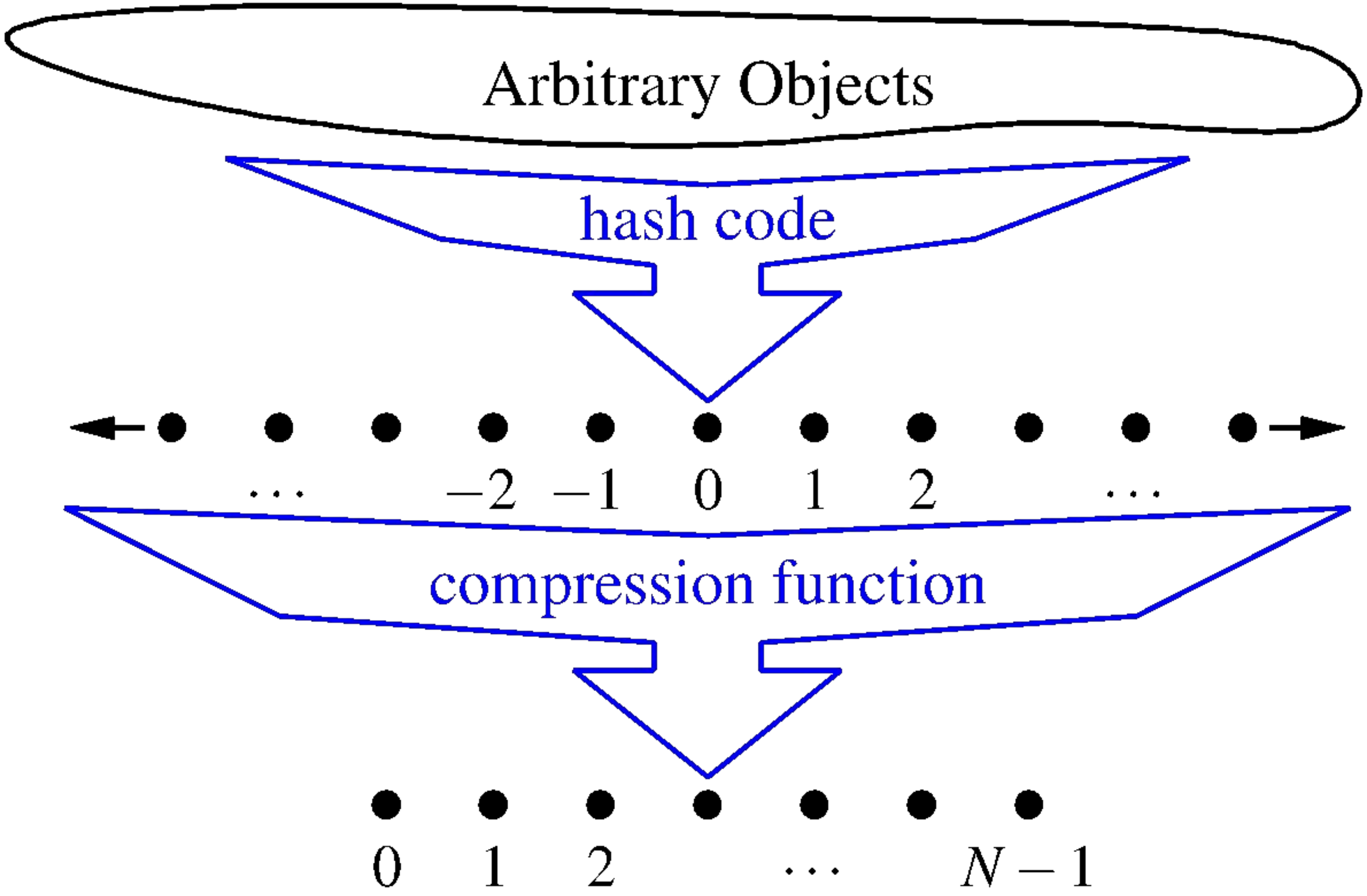
Hash Function Illustration



Hash Function Illustration



Hash Function Illustration



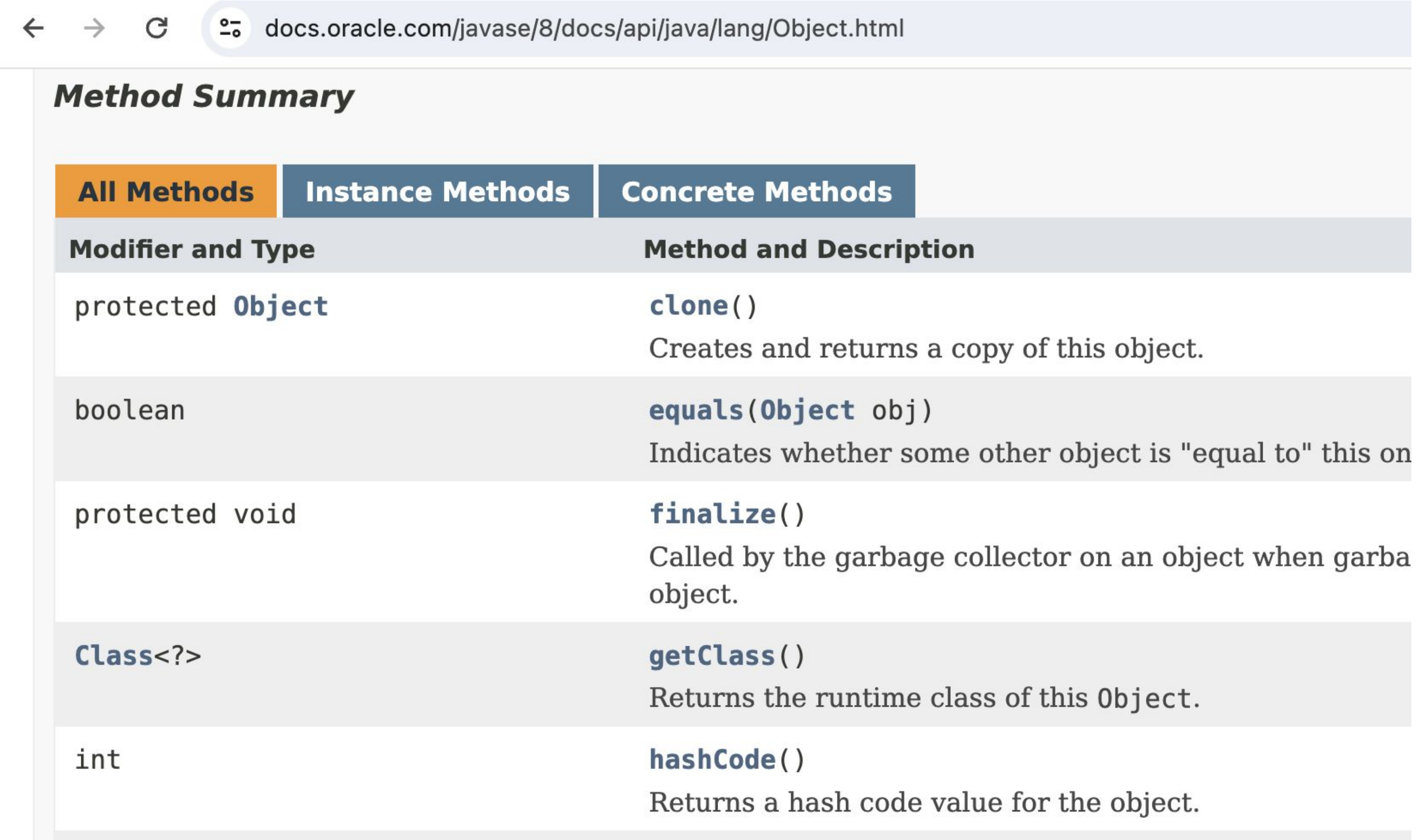
Hash Codes (h_1)

- 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**



Method Summary

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this on
`protected void`	`finalize()` Called by the garbage collector on an object when garba object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` 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: <u>M</u>ultiply <u>A</u>dd and <u>D</u>ivide (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.hashCode()` // 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!

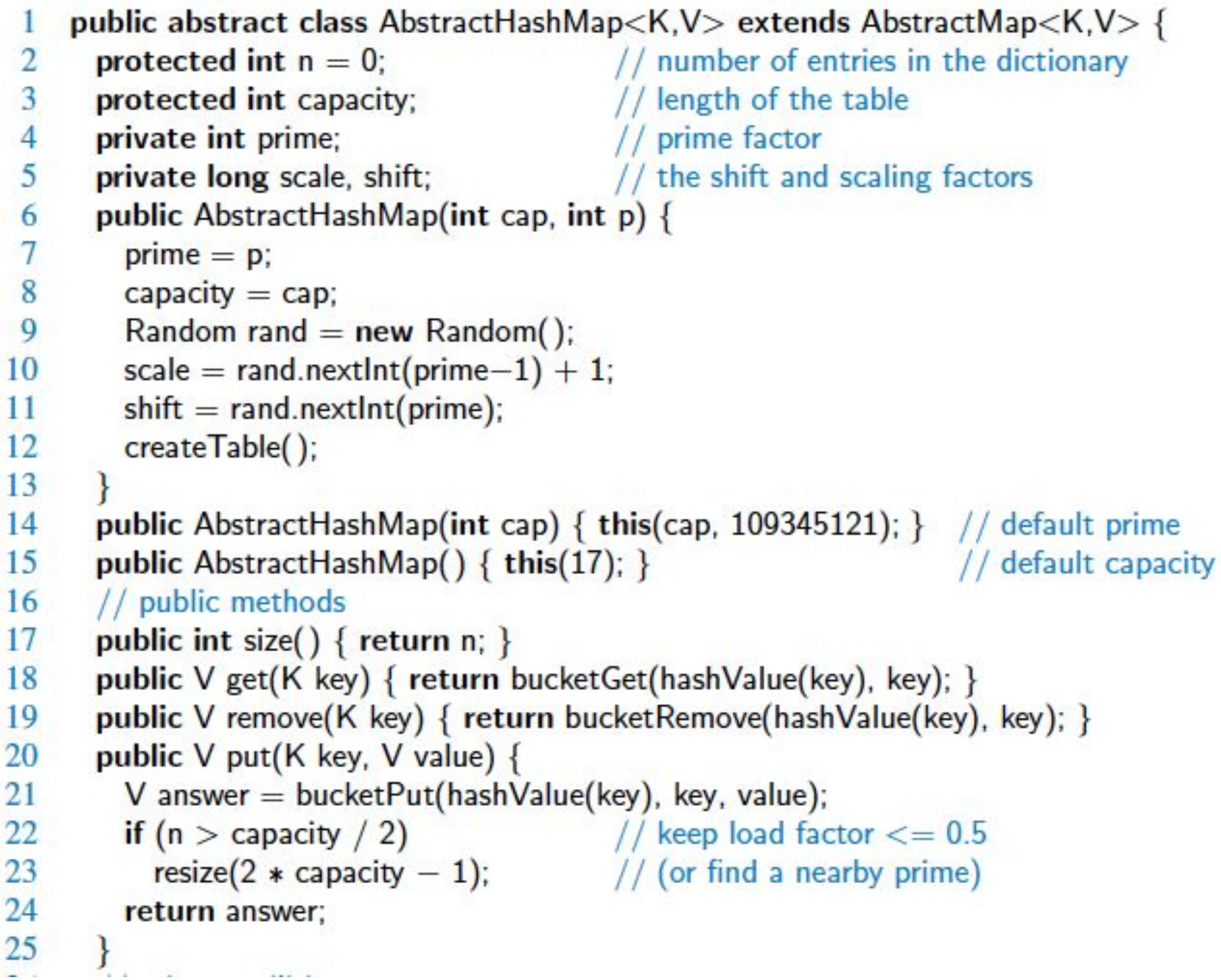
What should we do if there’s a collision?

- For a first impl, let’s just overwrite

Performance Analysis

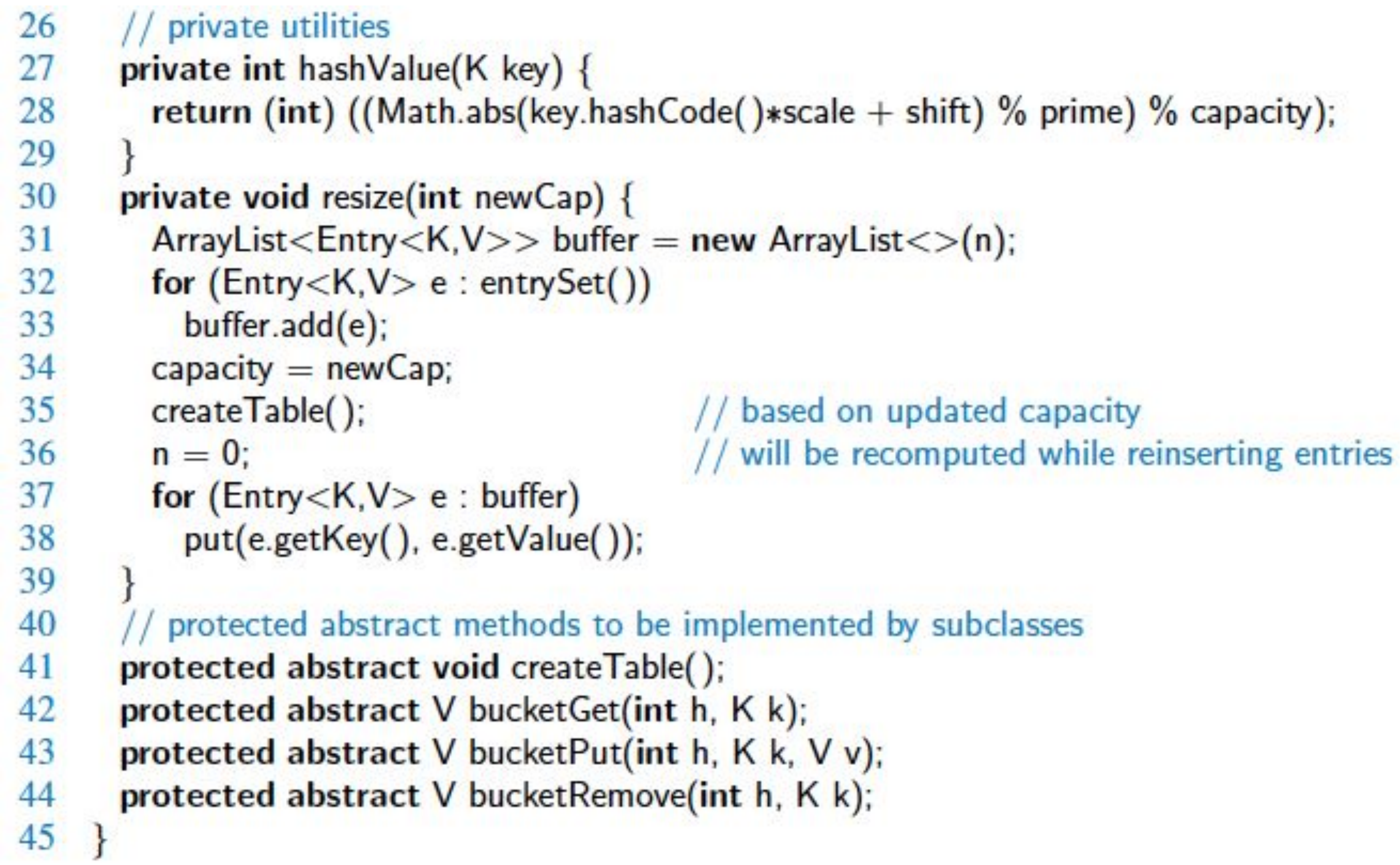
	ArrayMap	Collision Resistant Hash Map
get		
put		
remove		

AbstractHashMap



```
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;                   // number of entries in the dictionary
  protected int capacity;                // length of the table
  private int prime;                     // prime factor
  private long scale, shift;             // the shift and scaling factors
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random();
    scale = rand.nextInt(prime−1) + 1;
    shift = rand.nextInt(prime);
    createTable();
  }
  public AbstractHashMap(int cap) { this(cap, 109345121); }   // default prime
  public AbstractHashMap() { this(17); }                      // default capacity
  // public methods
  public int size() { return n; }
  public V get(K key) { return bucketGet(hashValue(key), key); }
  public V remove(K key) { return bucketRemove(hashValue(key), key); }
  public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key, value);
    if (n > capacity / 2)                // keep load factor <= 0.5
      resize(2 ∗ capacity − 1);          // (or find a nearby prime)
    return answer;
  }
```

AbstractHashMap



```
  // private utilities
  private int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
  }
  private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
    for (Entry<K,V> e : entrySet())
      buffer.add(e);
    capacity = newCap;
    createTable();                   // based on updated capacity
    n = 0;                           // will be recomputed while reinserting entries
    for (Entry<K,V> e : buffer)
      put(e.getKey(), e.getValue());
  }
  // protected abstract methods to be implemented by subclasses
  protected abstract void createTable();
  protected abstract V bucketGet(int h, K k);
  protected abstract V bucketPut(int h, K k, V v);
  protected abstract V bucketRemove(int h, K k);
}
```

Handling Collisions

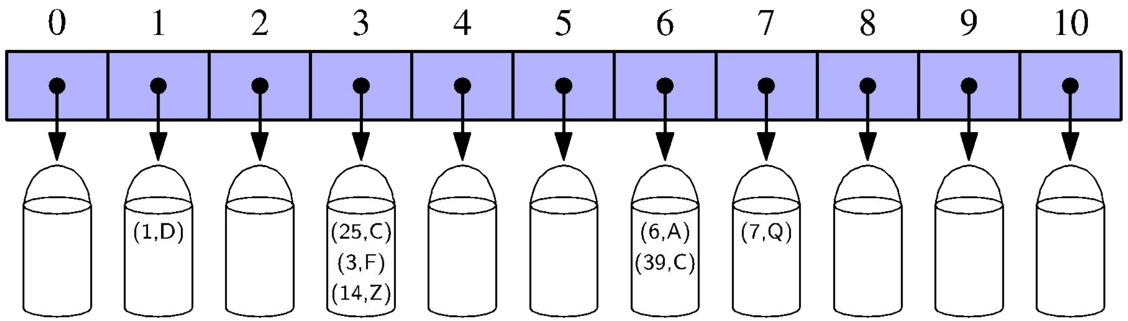
Handling Collisions

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

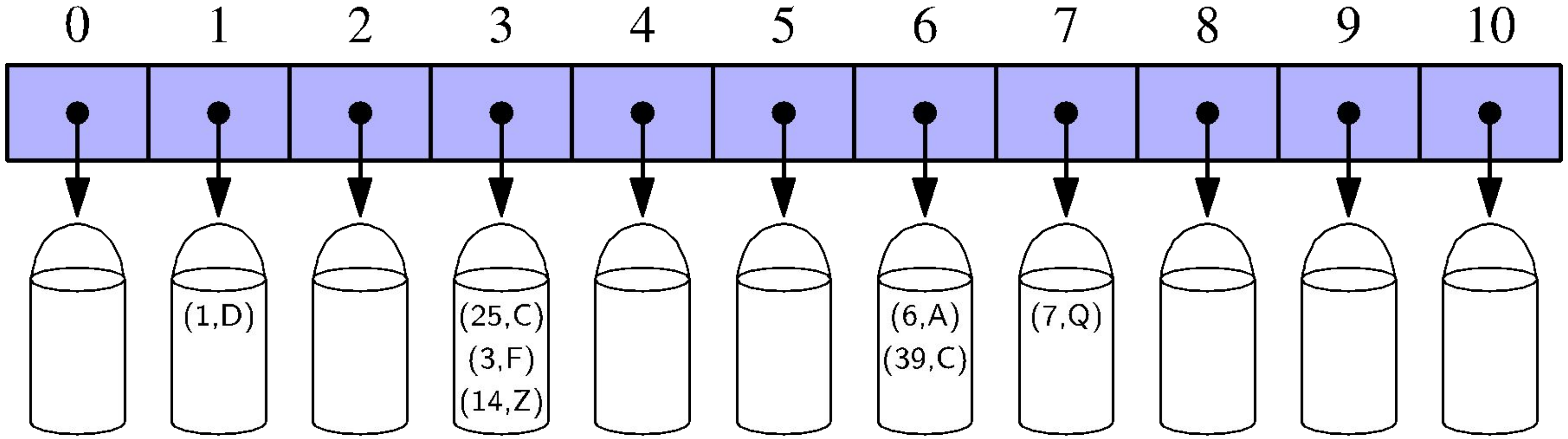
Separate Chaining: let each cell in the table point to a linked list of entries that map there

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 h1 and h2
- h2 compresses output of h1 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*"

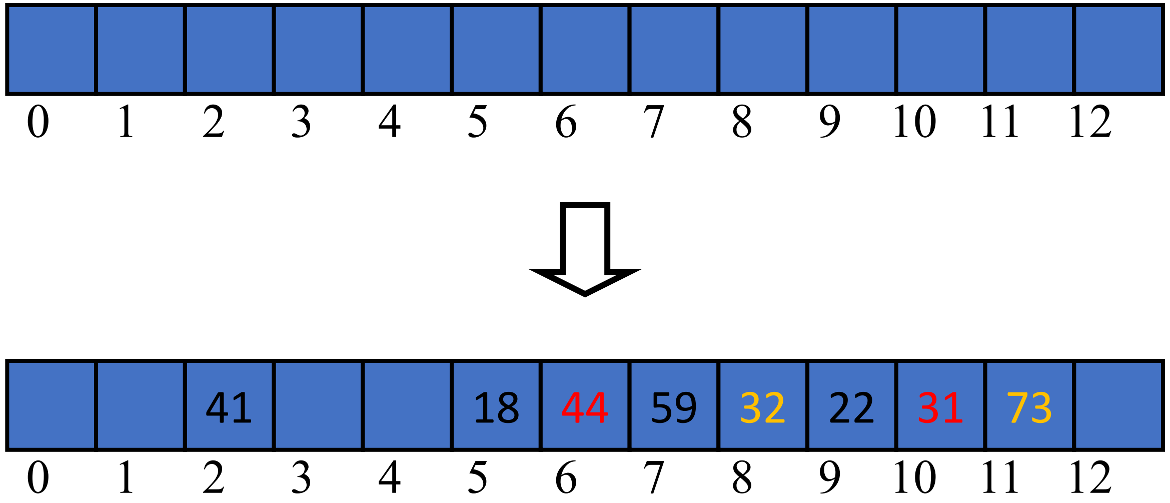
(h(k)+1)%n

(h(k)+2)%n

….

(h(k)+i)%n

until you find an empty slot!



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

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

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
  ....
  (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

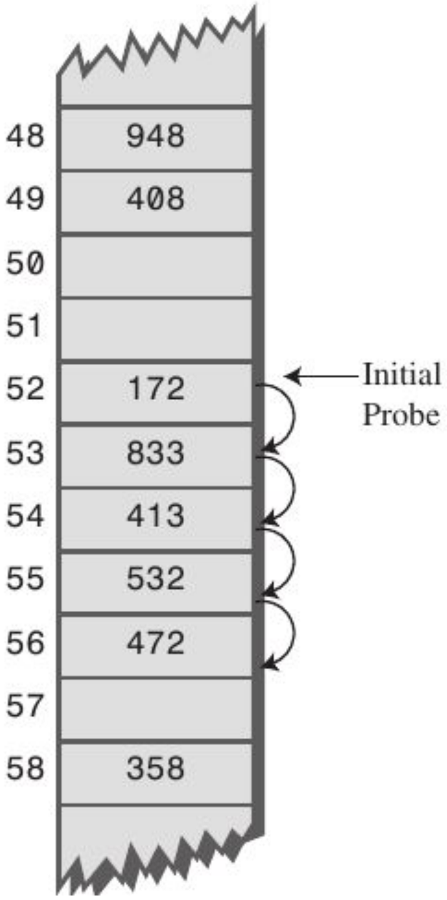
  (h(k)+f(2)) % n

  ….

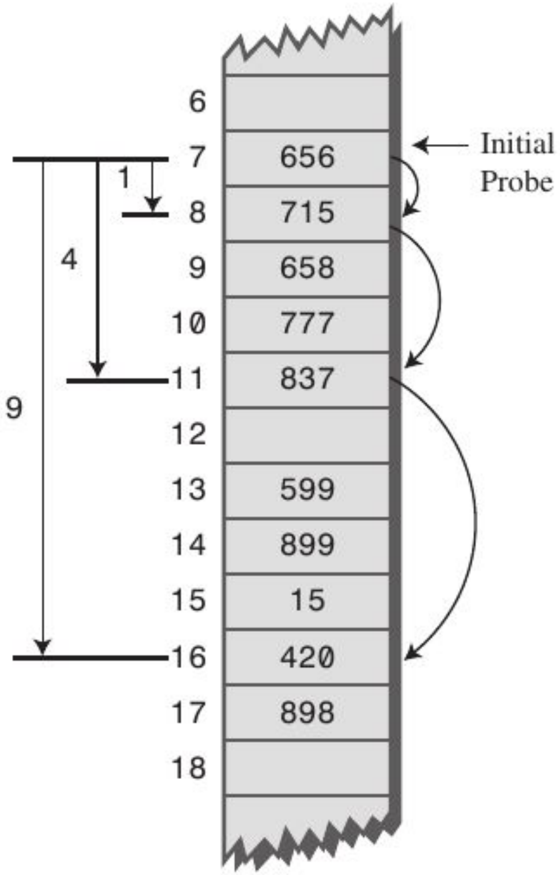
  (h(k)+f(i)) % n

  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

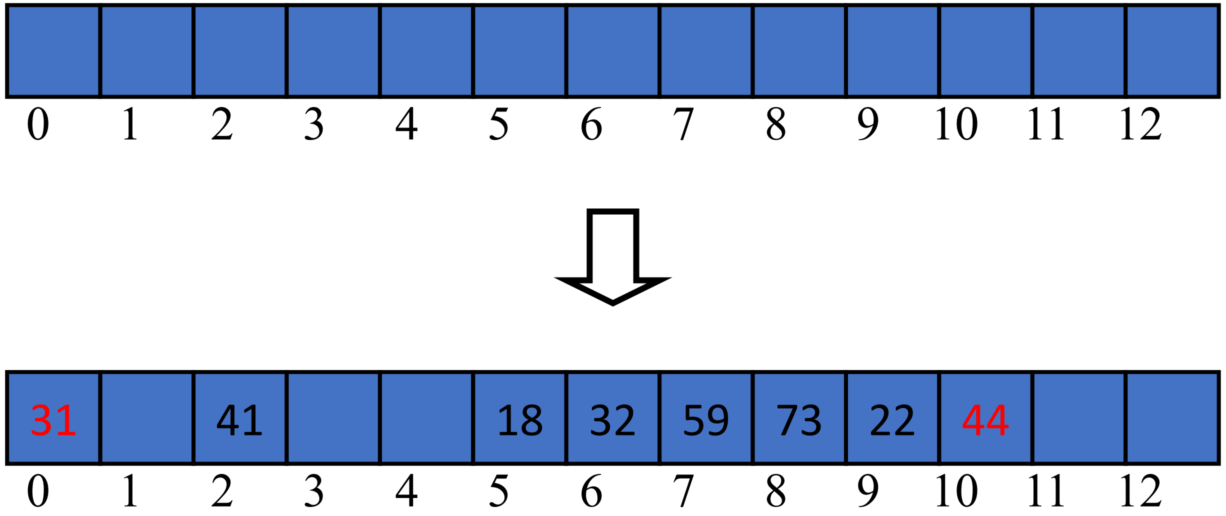
probe:

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

h(k) = k % 13

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

h'(k) = 7 - k % 7



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

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

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

Summary

Maps:

- Associative key-value pairs
- O(n) operations if implemented with an array

Hash functions:

- one-way collision resistant functions
- map an input to a fixed length output

Hash Maps:

- Use hash function to index - Removes need for looping
- When collisions occur we can handle it with chaining or open addressing