CS151 Intro to Data Structures

Hashmaps

Announcements & Outline

HW6 and Lab8 due last night

Next homework (HashMaps and QuickSort) due April 18th

Only two more homeworks left! 2 week timeline for each

Today:

- Maps and Hash Functions review
- Hash Maps - a magical data structure
- Collision handling strategies

Maps

- Data Structure that stores a collection of key-value pairs
- Implementation: ArrayMap of SimpleEntry
- What is a SimpleEntry?
- How did we implement these and what is the runtime complexity?
  - put
  - get
  - remove

HashMaps

- A hash function maps an arbitrary length input to a fixed length *unique* output

- Properties of a good hash function:
  - one way
  - **collision resistant**
  - uniformity
  - quick to compute

- Improves upon ArrayMap implementation by using the hash function output as our array index

A Simple Hash Function

h(x) = x % N

How would the following operations look?

- put
- get
- remove

A Better Hash Function...

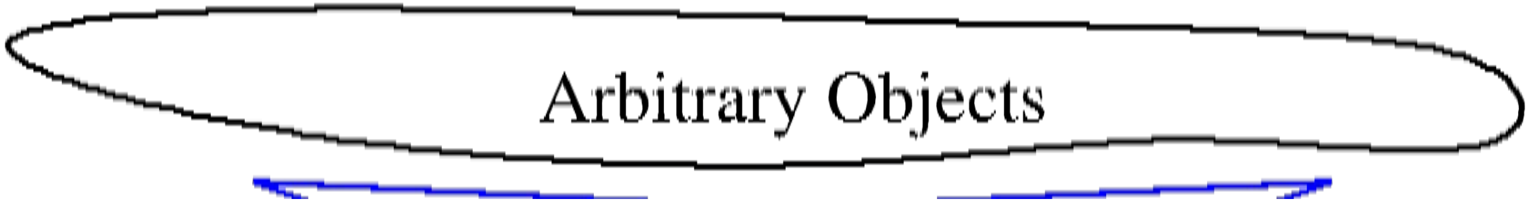
Problems with our hash: $h(x) = x \% N$

1. Collisions!
2. What if the key x is not an integer?

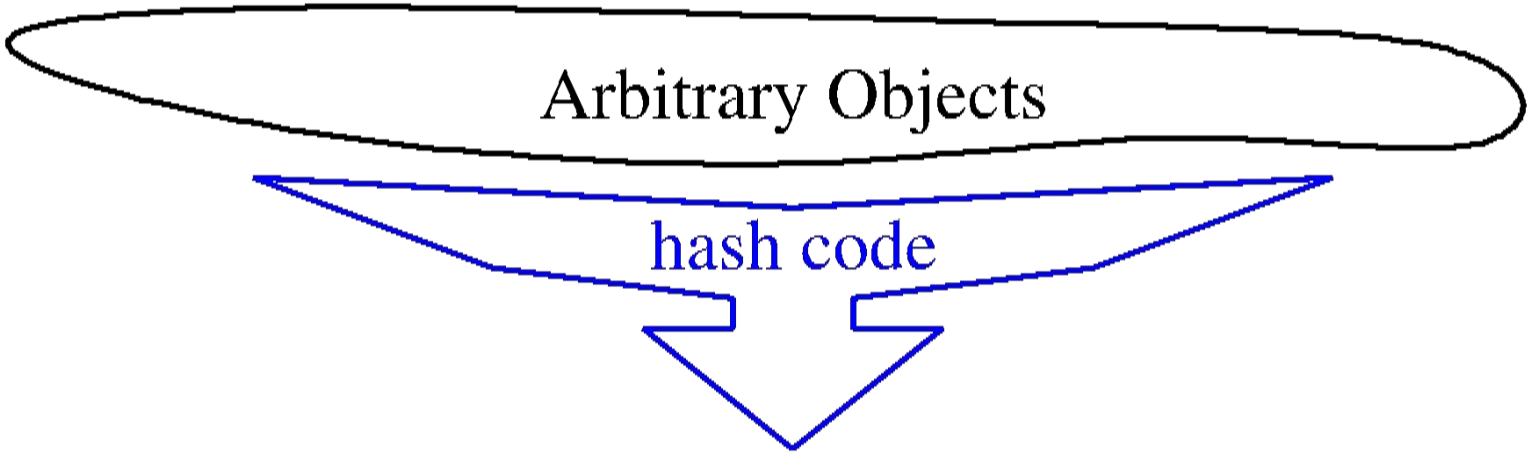
Solution:

- Use a composition of two hash functions h1 and h2
- h1(x) maps keys to integers
- h2(x) "compresses" the output in a range from [0, N) to fit as array indices
- h(x) = h2(h1(x))

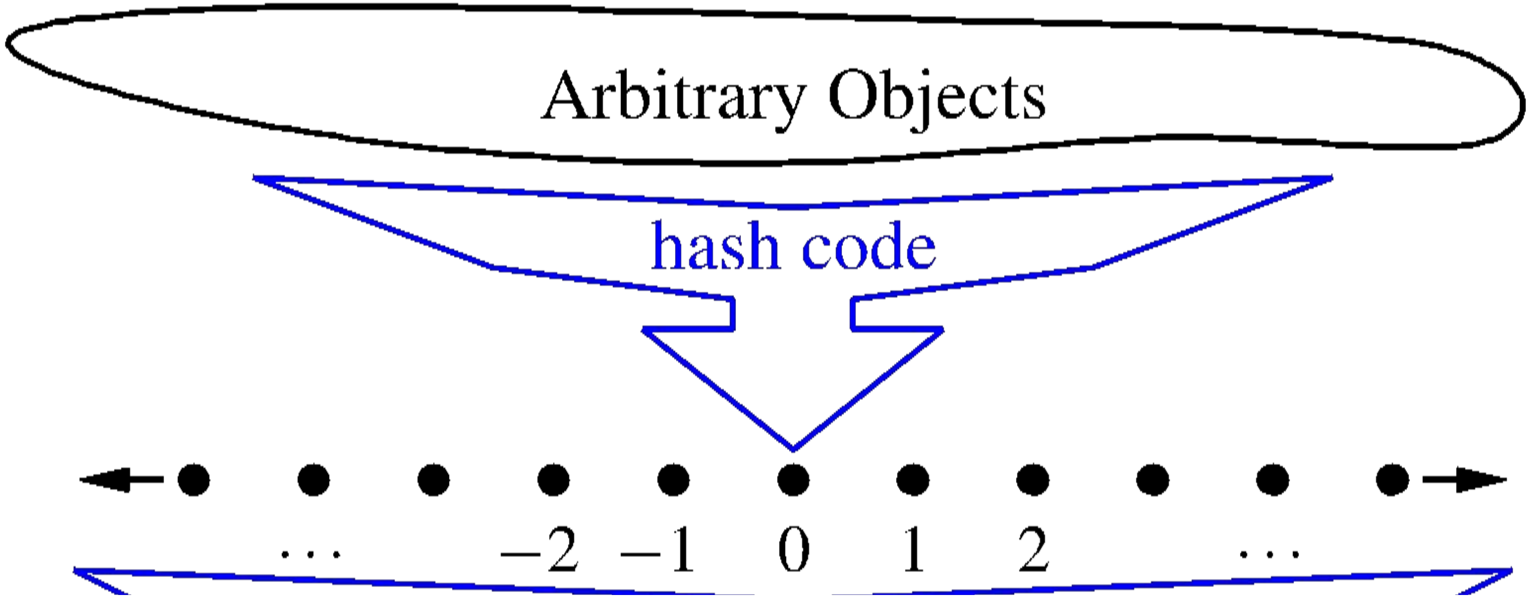
Hash Function Illustration



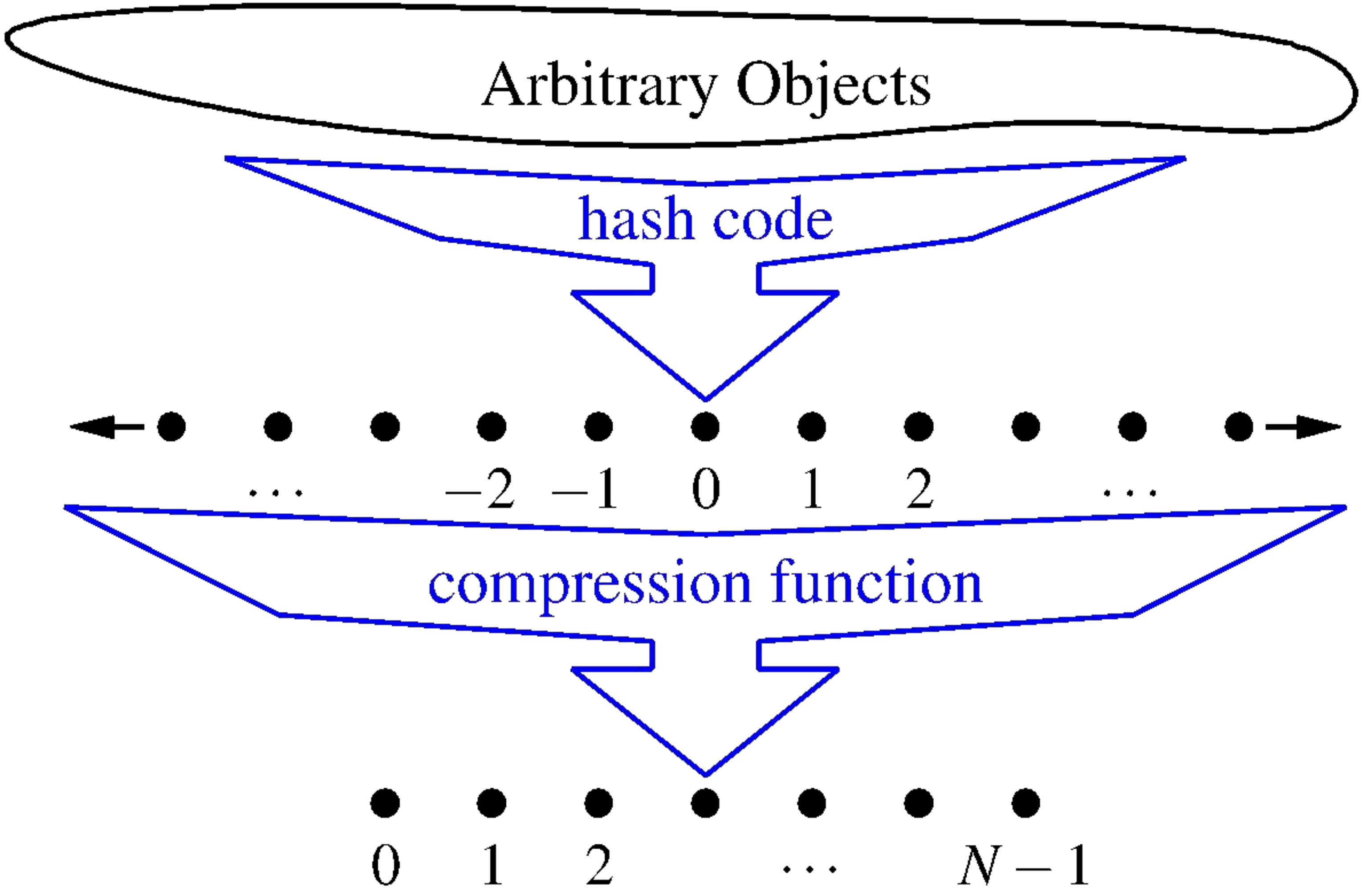
Hash Function Illustration



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

Solution: Memory Addresses

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

docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

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

Putting them together...

Book's `AbstractHashMap` hash method uses:

h_1`(k) = k.hashCode()` // java memory address

h_2`(x) = ((ax + b) % p) % N`

`h = h2(h1(k))`

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

```java
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

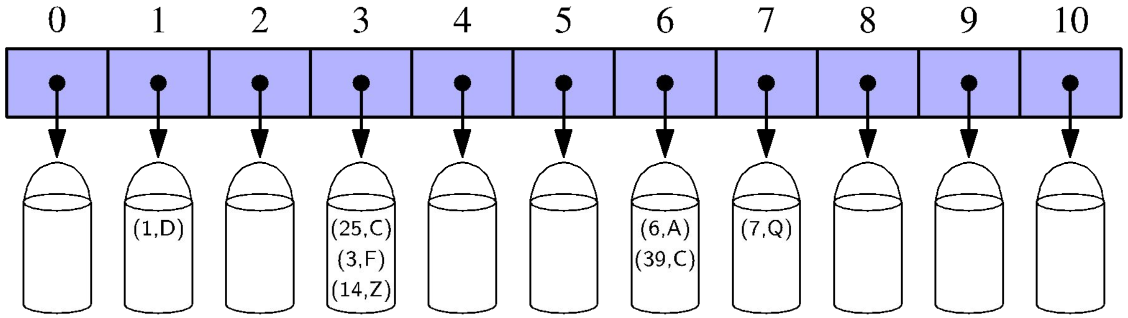
Handling Collisions

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

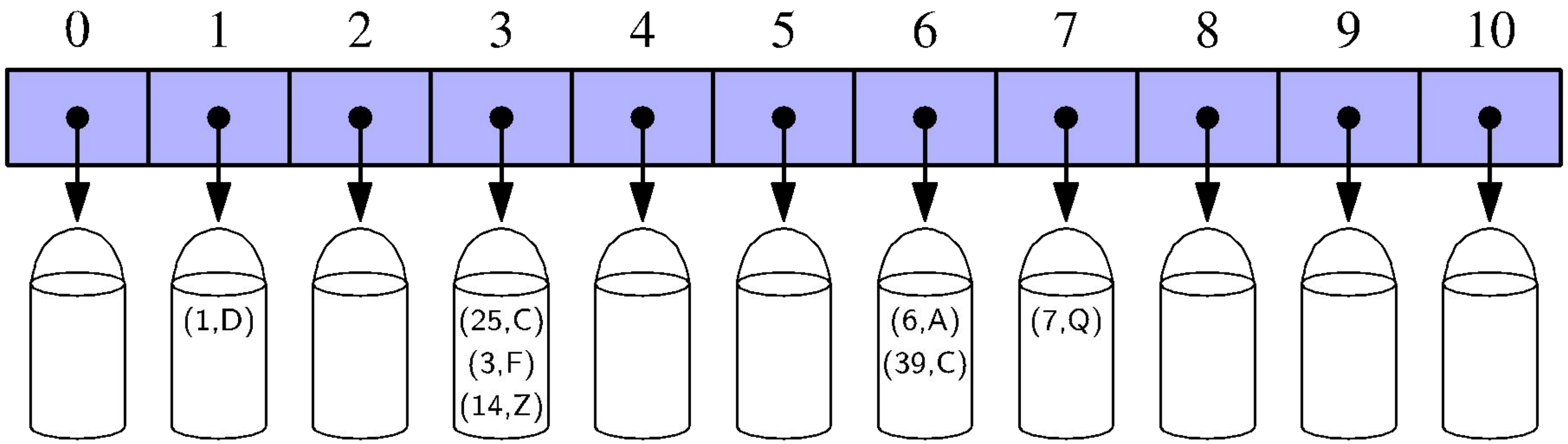
Separate Chaining: let each cell in the table point to a linked list of entries that map there

Simple, but requires additional memory besides the table

Let's implement a ChainHashMap

What data structure should we use for the buckets?

- LinkedList!



Collision Handling Approach #2

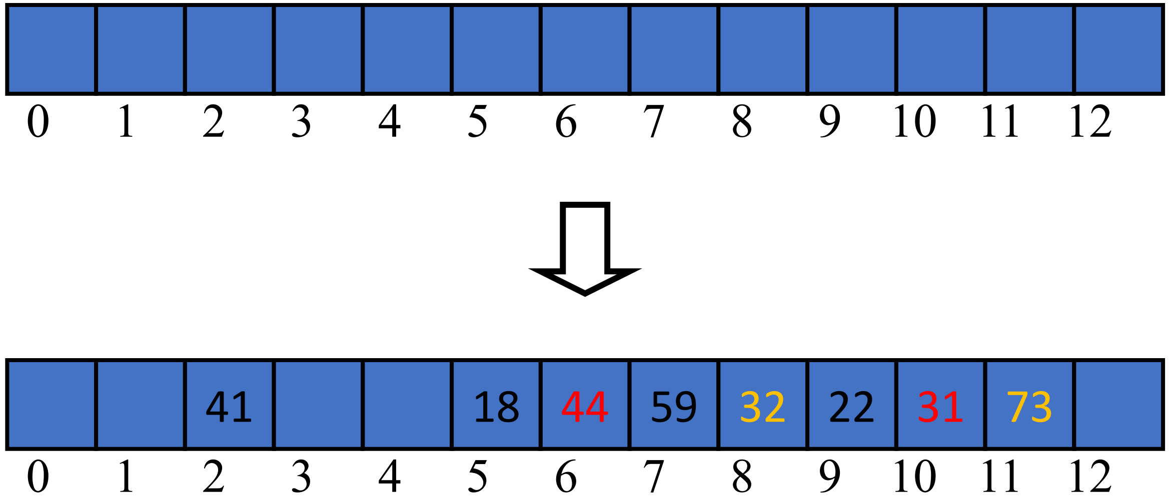
Open Addressing and Probing

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 look at an implementation of ProbeHashMap

Open Addressing and Probing

Linear Probing (what we just saw):

- 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

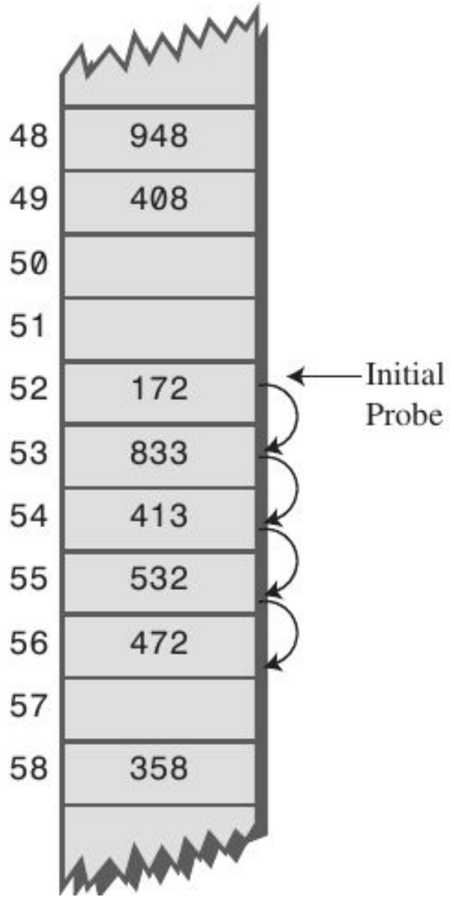
  (h(k)+f(2)) % n

  ....

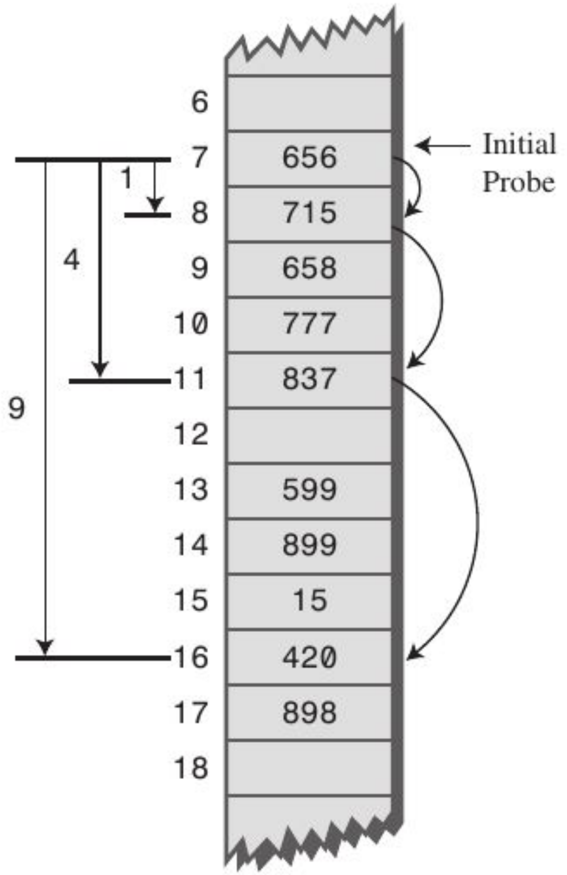
  (h(k)+f(i)) % n

  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

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 strategies:

- Chaining: use a LL
- Probing: use a secondary hash function