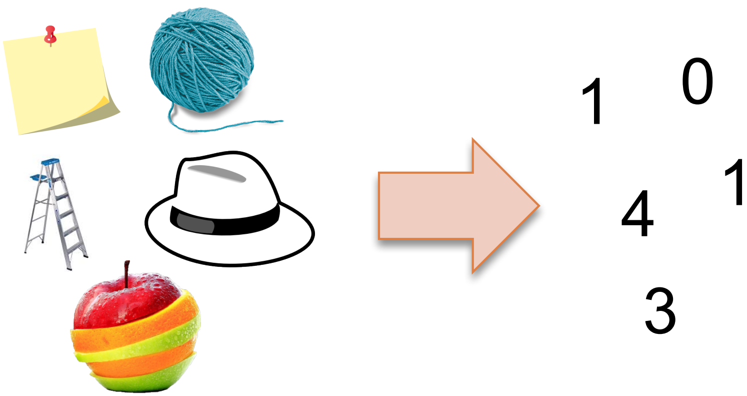
HASHING II

CS2110
Spring 2018

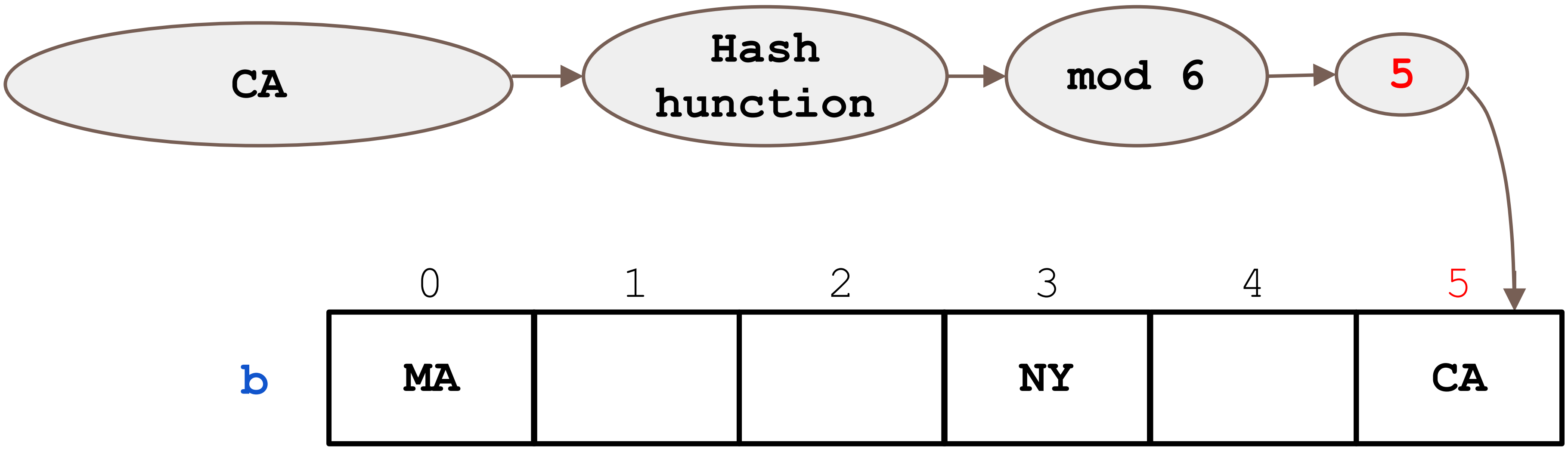
Hash Functions



- Requirements:
  1) deterministic
  2) return a number in [0..n]
- Properties of a good hash:
  1) fast
  2) collision-resistant
  3) evenly distributed
  4) hard to invert

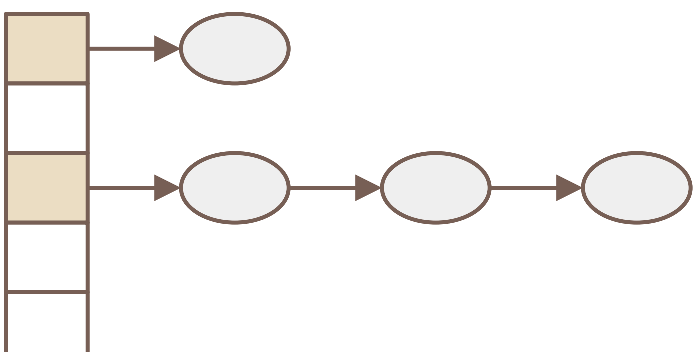
Hash Table

add("CA")

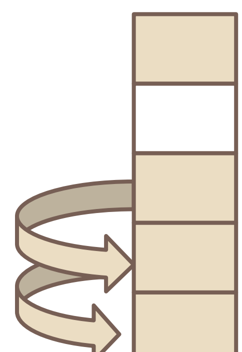


Two ways of handling collisions:

1. Chaining



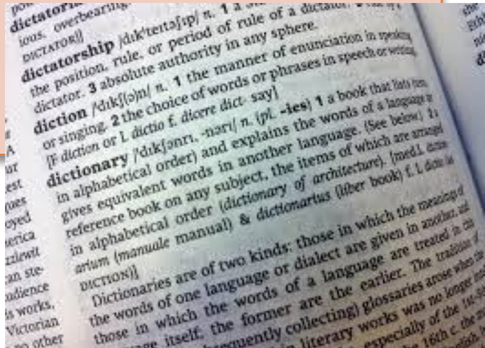
2. Open Addressing



HashSet and HashMap

```
Set<V>{

  boolean add(V value);

  boolean contains(V value);

  boolean remove(V value);

}
```

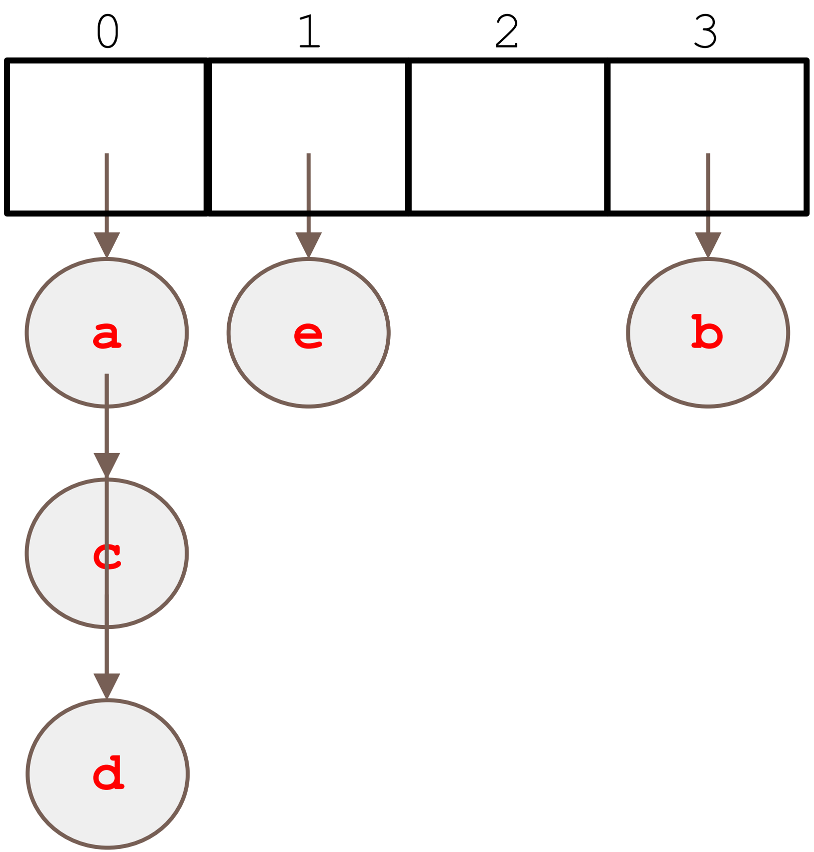
```
Map<K,V>{

  V put(K key, V value);

  V get(K key);

  V remove(K key);

}
```



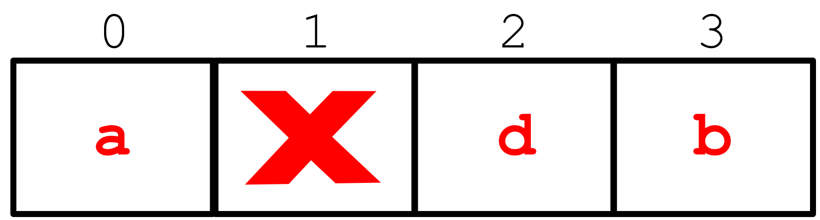
Remove

```
put('a')
put('b')
put('c')
put('d')
get('d')
remove('c')
get('d')
put('e')
```

Chaining



Open Addressing



Time Complexity (no resizing)

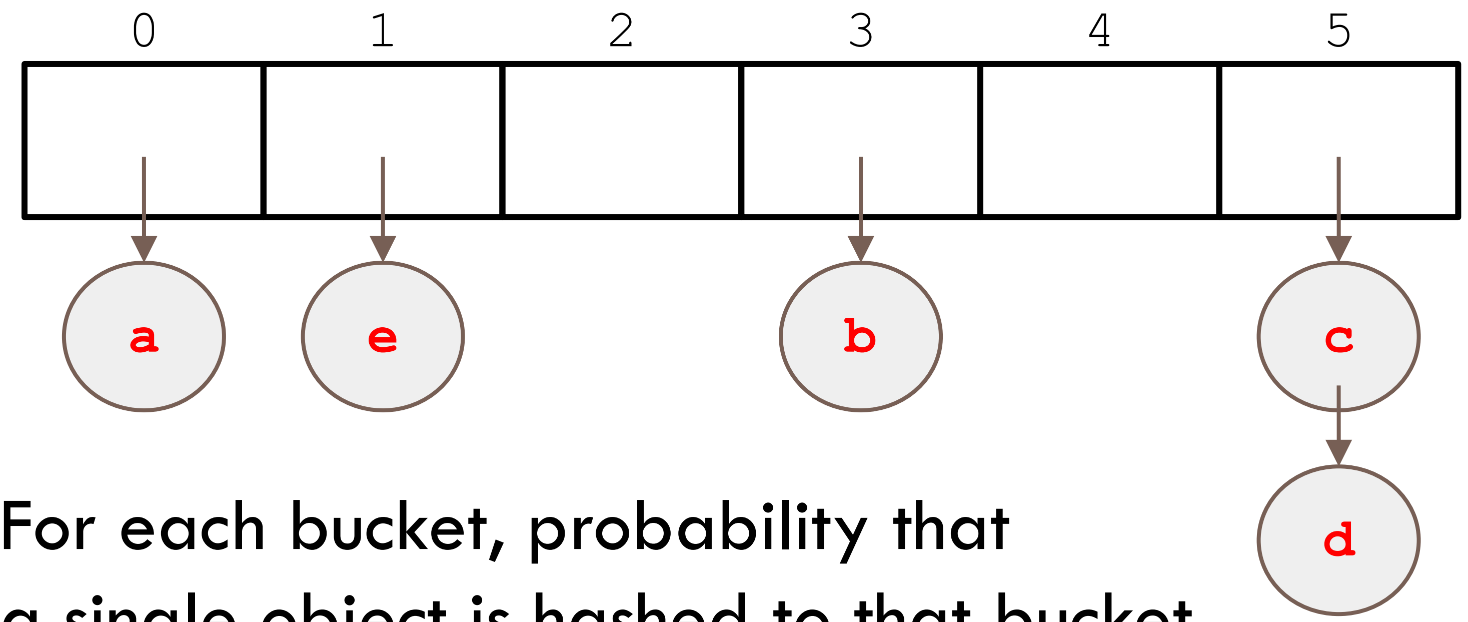
Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(n)$	$O(n)$
Open Addressing	$O(n)$	$O(n)$	$O(n)$

Load Factor

Load factor

$$\lambda = \frac{\textit{\# of entries}}{\textit{length of array}}$$

Expected Chain Length



- For each bucket, probability that a single object is hashed to that bucket is 1/length of array
- There are n objects in the hash table
- Expected length of chain is n/length of array = λ

Expected Time Complexity (no resizing)

Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(1 + \lambda)$	$O(1 + \lambda)$
Open Addressing			

Expected Number of Probes

0	1	2	3	4	5

- We always have to probe H(v)
- With probability λ, first location is full, have to probe again
- With probability $\lambda \cdot \lambda$, second location is also full, have to probe yet again
- …
- Expected #probes $= 1 + \lambda + \lambda^2 + \ \ldots = \frac{1}{1-\lambda}$

Expected Time Complexity (no resizing)

Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(1)$	$O(1)$
Open Addressing	$O(1)$	$O(1)$	$O(1)$

Assuming constant load factor

We need to dynamically resize!

Amortized Analysis

vs.

- In an **amortized analysis**, the time required to perform a sequence of operations is averaged over all the operations
- Can be used to calculate **average cost** of operation

Amortized Analysis of put

- Assume dynamic resizing with load factor $\lambda = \frac{1}{2}$:
  - Most put operations take (expected) time $O(1)$
  - If $i = 2^j$, put takes time $O(i)$
  - Total time to perform n put operations is $n \cdot O(1) + O(2^0 + 2^1 + 2^2 + \ ... + 2^j)$
  - Average time to perform 1 put operation is $O(1) + O\left(\frac{1}{2^j} + \frac{1}{2^{j-1}} + \ ... + \frac{1}{4} + \frac{1}{2} + 1\right) = O(1)$

Expected Time Complexity (with dynamic resizing)

Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(1)$	$O(1)$
Open Addressing	$O(1)$	$O(1)$	$O(1)$

Cuckoo Hashing

Cuckoo Hashing

- Alternative solution to collisions
- Assume you have two hash functions H1 and H2

element	a	b	c	d	e
H1	0	9	17	11	5
H2	5	2	10	3	13

0	1	2	3	4	5
a		b	d b	c	e d

What if there are loops?

Complexity of Cuckoo Hashing

- Worst Case:

Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(n)$	$O(n)$
Open Addressing	$O(n)$	$O(n)$	$O(n)$
Cuckoo Hashing	∞	$O(1)$	$O(1)$

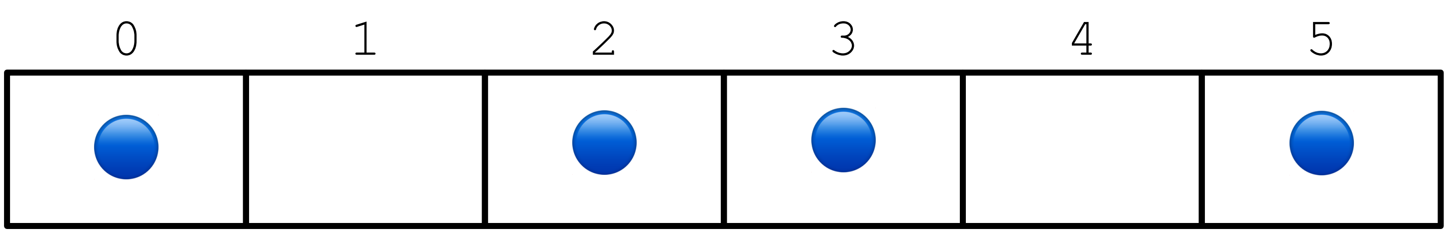
- Expected Case:

Collision Handling	put(v)	get(v)	remove(v)
Chaining	$O(1)$	$O(1)$	$O(1)$
Open Addressing	$O(1)$	$O(1)$	$O(1)$
Cuckoo Hashing	$O(1)$	$O(1)$	$O(1)$

Bloom Filters

- Assume we only want to implement a set
- What if you had stored the value at "all" hash locations (instead of one)?

element	a	b	c	d	e
H1	0	9	17	11	5
H2	5	2	10	3	13



Features of Bloom Filters

- Worst-case $O(1)$ put, get, and remove
- Works well with higher load factors
- But: false positives