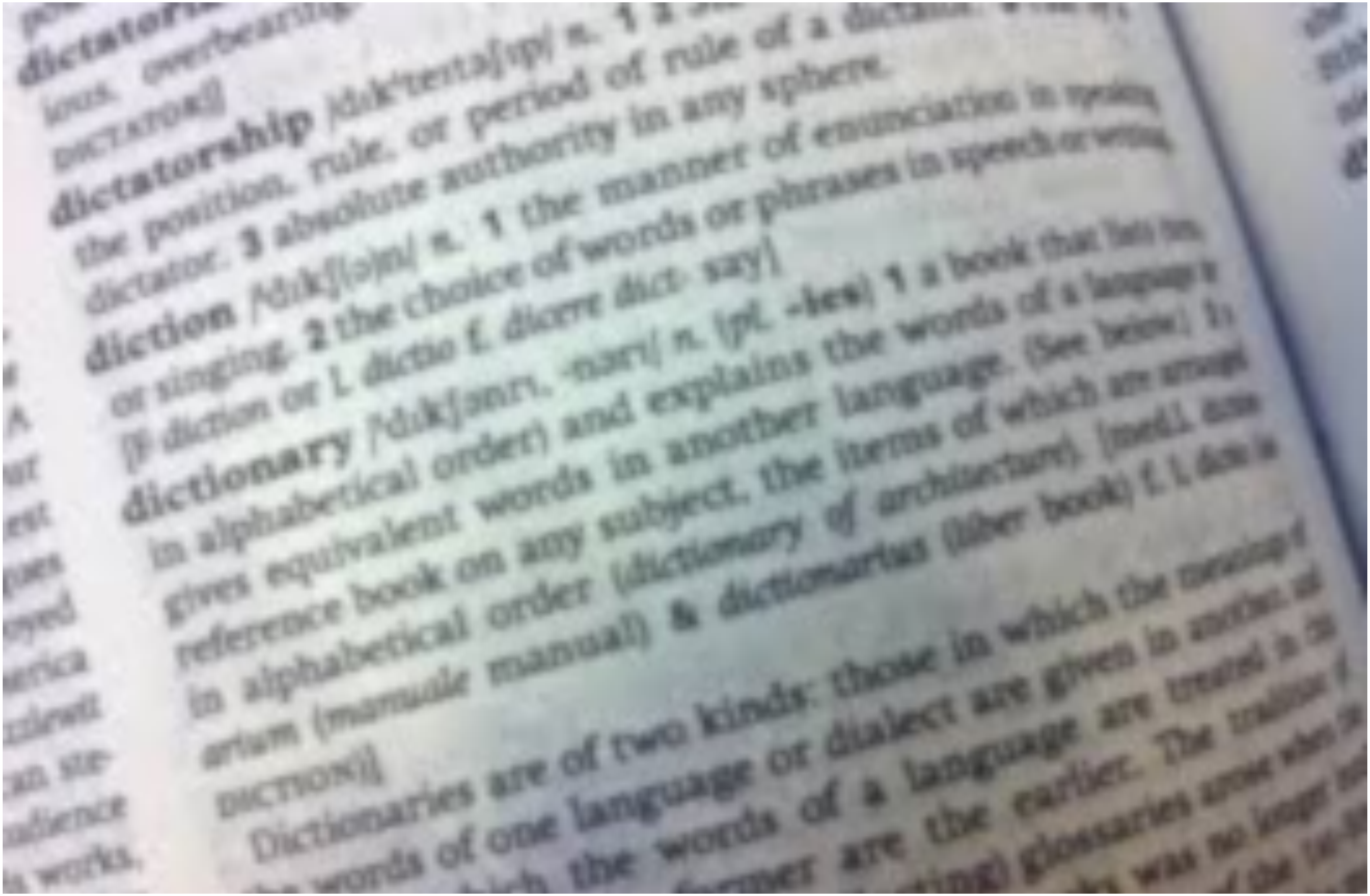


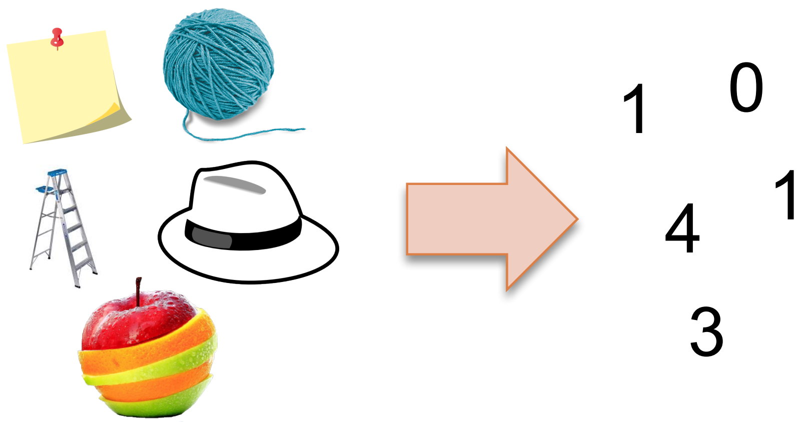
HASHING

CS2110
Spring 2018

Announcements

- Submit Prelim 2 conflicts by tomorrow night
- A7 Due FRIDAY
- A8 will be released on Thursday

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 Functions

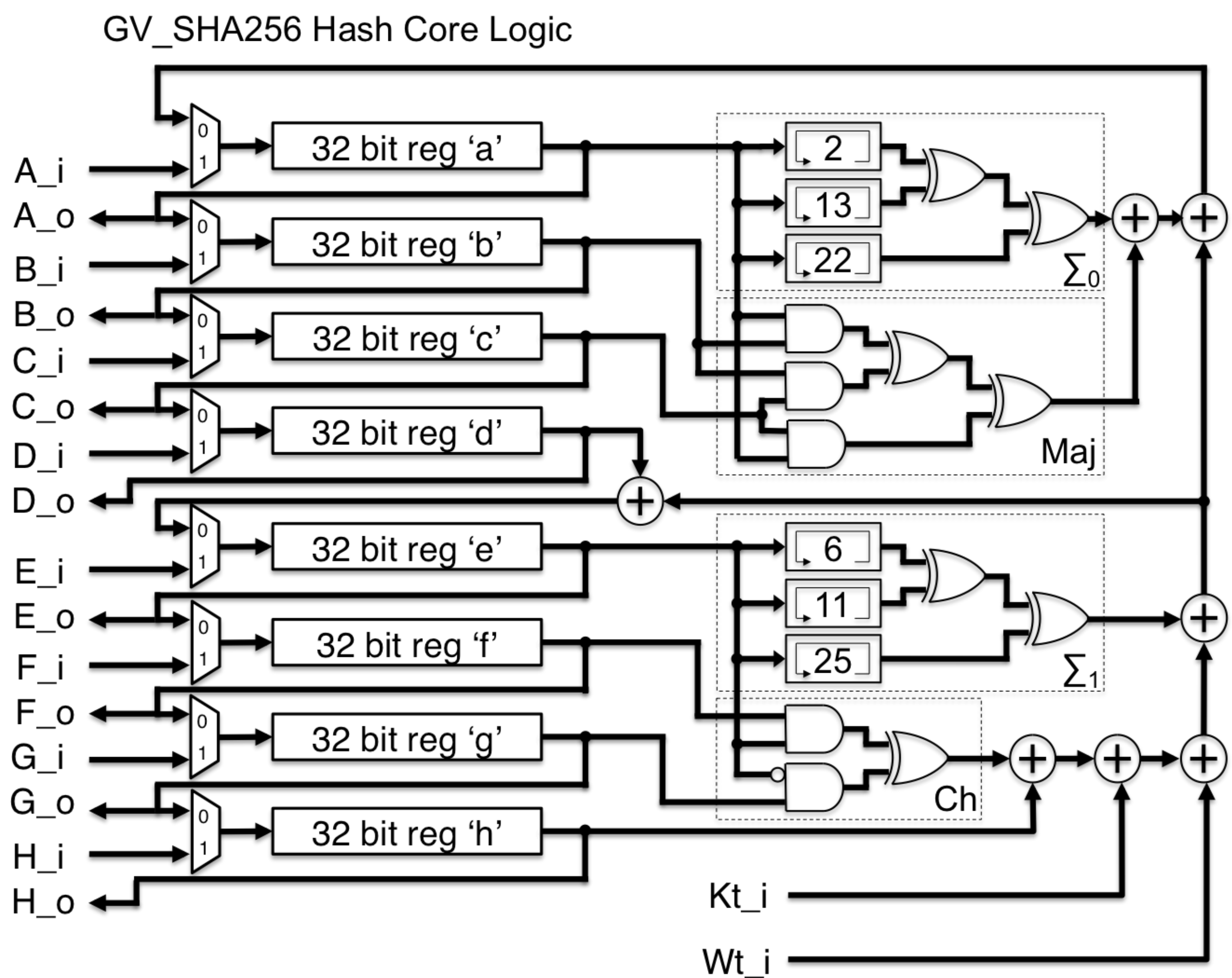
1 0 4 1 3

- Requirements:
  1) deterministic
  2) return a number in [0..n]

Which of the following functions f: Object -> int are hash functions:

a) f(x) = x
b) f(x) = x.hashCode()
c) f(x) = &x
d) f(x) = 0

Example: SHA-256



Example: hashCode()

- Method defined in java.lang.Object
- Default implementation: uses memory address of object
  - If you override equals, you must override hashCode!!!
- String overrides hashCode:

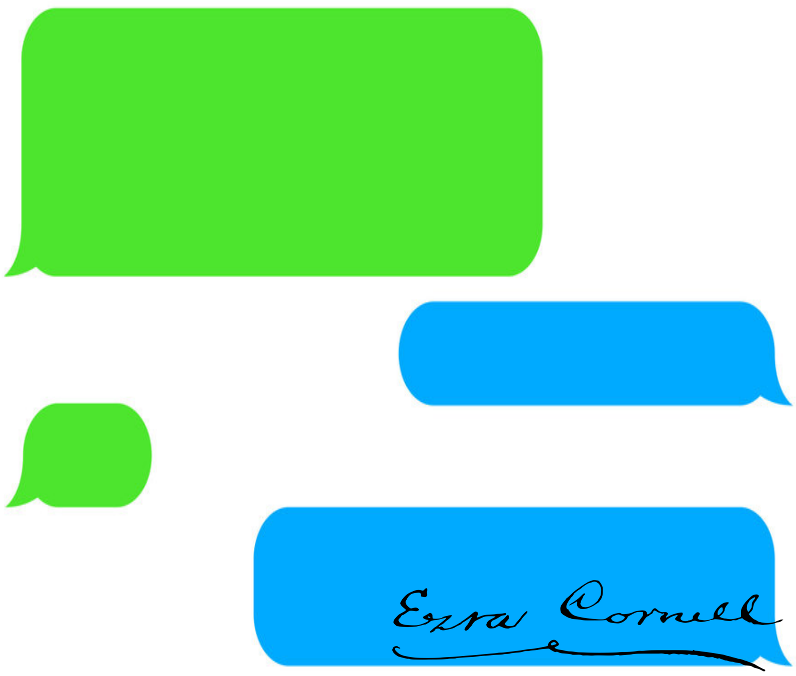
$$\mathrm{s.hashCode()} := s[0] * 31^{n-1} + s[1] * 31^{n-2} + \ldots + s[n-1]$$

Application: Error Detection

Submitted	Date	By	Size	MD5 What's this?
A6GUI	April 10, 2018 04:28PM	[illegible]	10.82 kB	ca62dd8fc1273f51baa6f507efac1d2b

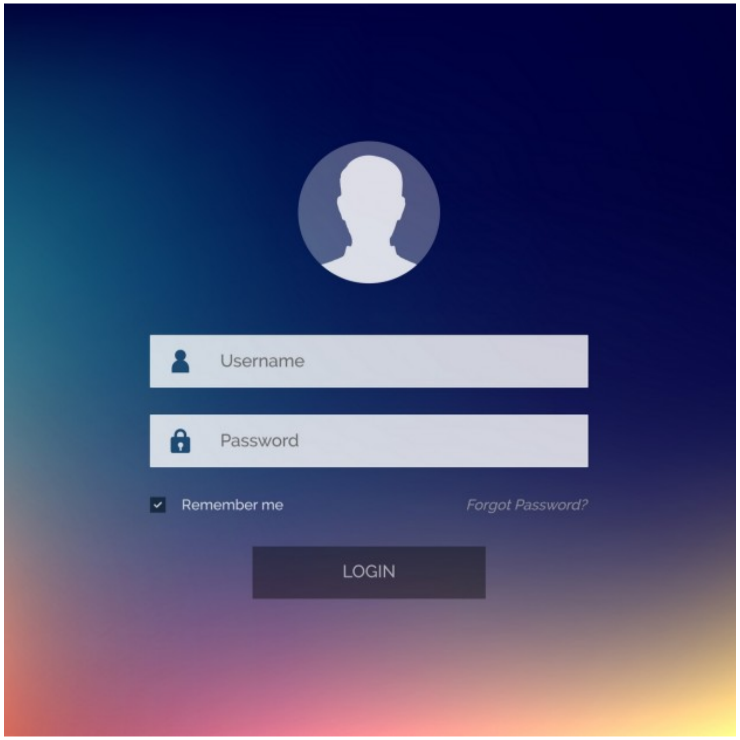
- Hash functions are used for error detection
- E.g., hash of uploaded file should be the same as hash of original file (if different, file was corrupted)

Application: Integrity



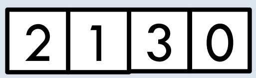
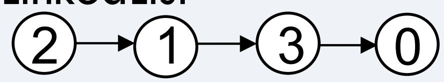
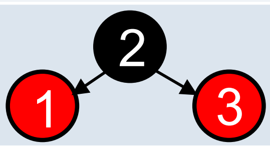
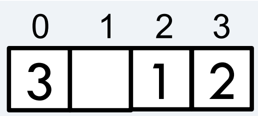
- Hash functions are used to "sign" messages
- Provides integrity guarantees in presence of an active adversary
- Principals share some secret sk
- Send (m, h(m,sk))

Application: Password Storage



- Hash functions are used to store passwords
- Could store plaintext passwords
  - Problem: Password files get stolen
- Could store (username, h(password))
  - Problem: password reuse
- Instead, store (username, s, h(password, s))

Application: Hash Set

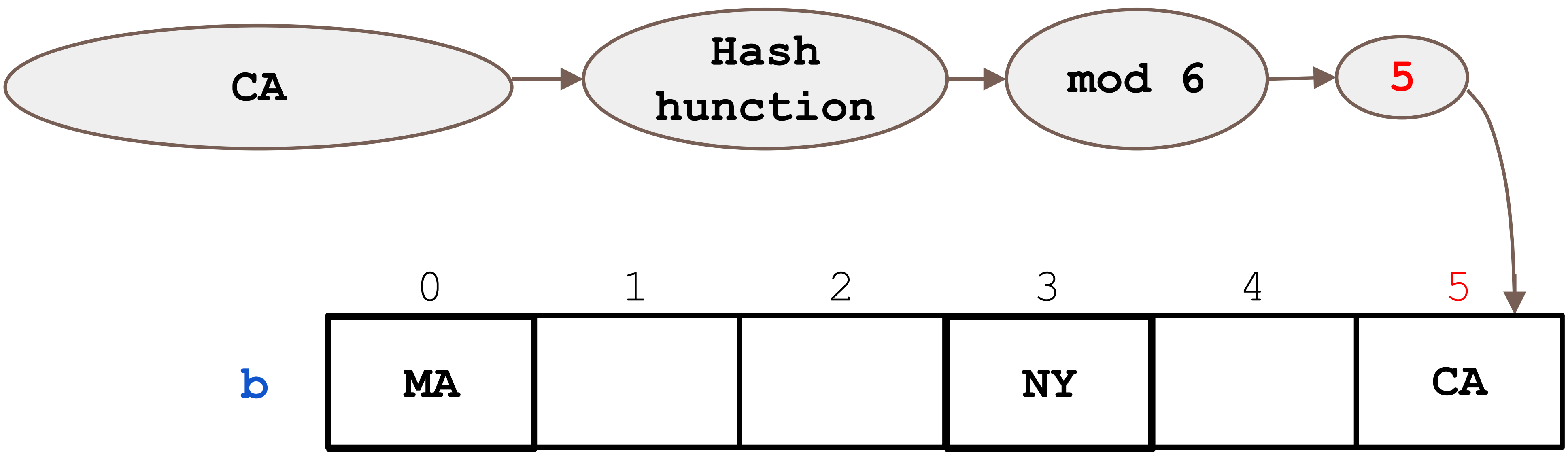
Data Structure	add(val x)	lookup(int i)	find(val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
TreeSet 	$O(\log n)$		$O(\log n)$
HashSet 	$O(1)$		$O(1)$

Expected time
Worst-case: $O(n)$

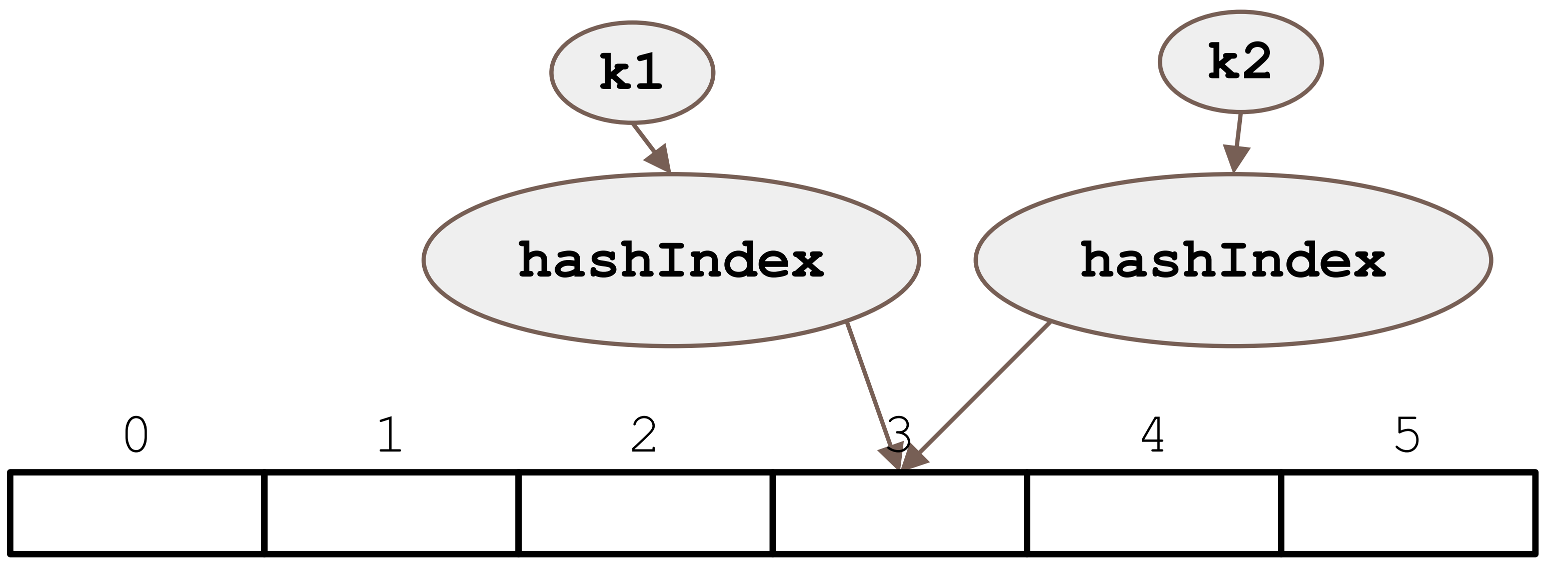
Hash Tables

Idea: finding an element in an array takes constant time when you know which index it is stored in

add("CA")



So what goes wrong?



Can we have perfect hash functions?

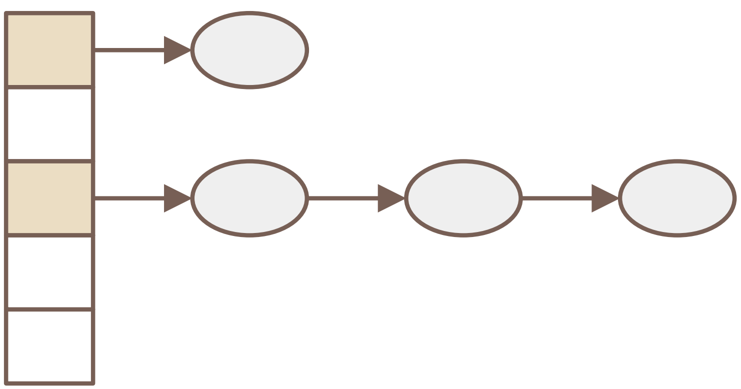
- Perfect hash functions map each value to a different index in the hash table

- Impossible in practice
  - don’t know size of the array
  - Number of possible values far far exceeds the array size
  - no point in a perfect hash function if it takes too much time to compute

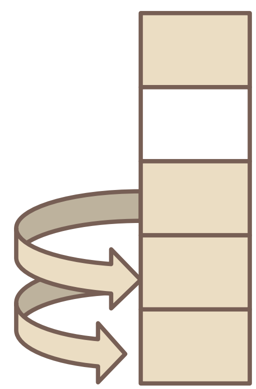
Collision Resolution

Two ways of handling collisions:

1. Chaining

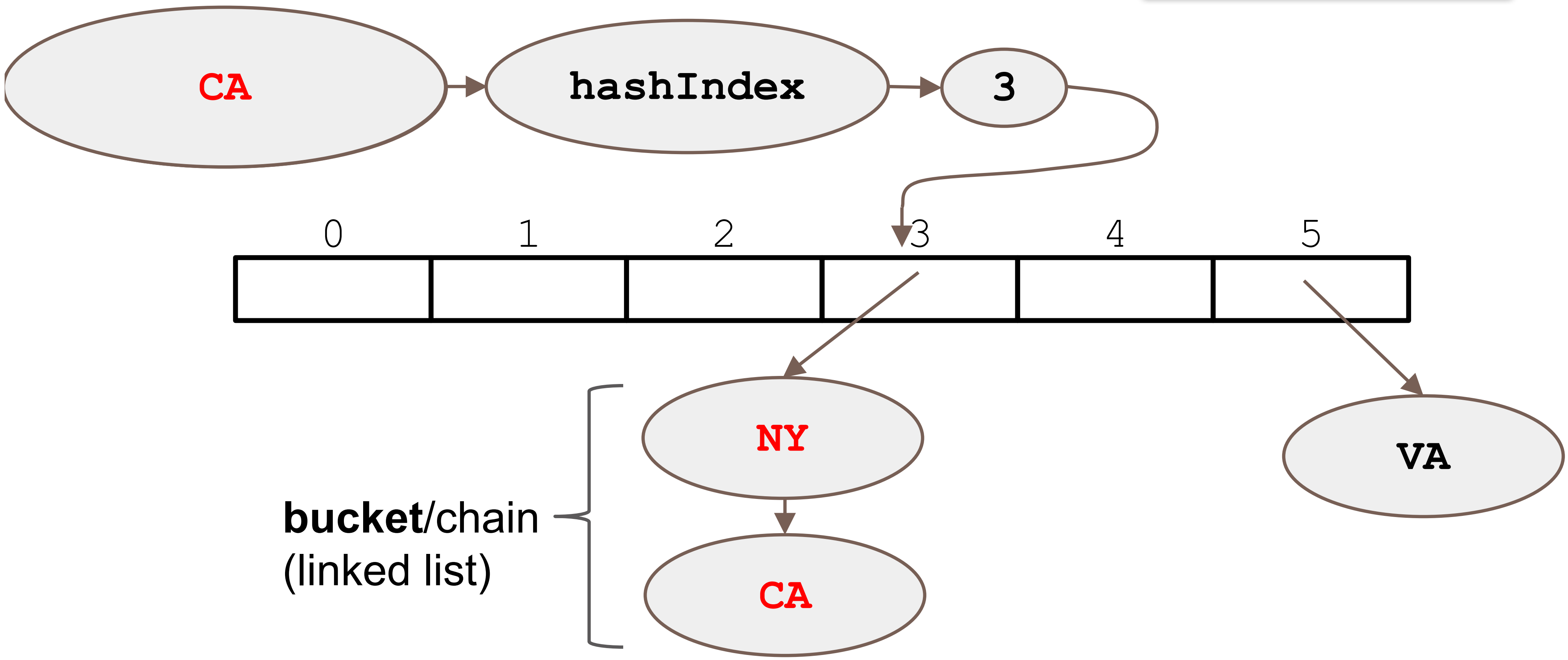


2. Open Addressing



Chaining

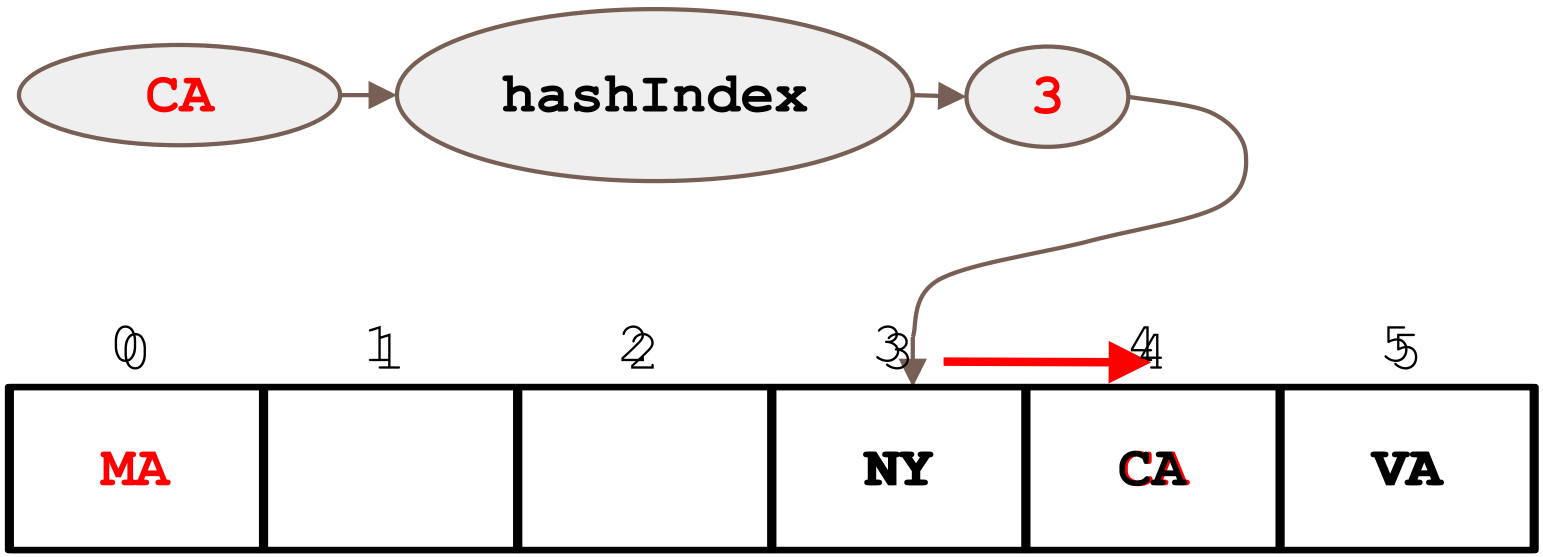
```
add("NY")
add("CA")
lookup("CA")
```



Open Addressing

probing: Find another available space

`add("CA")`



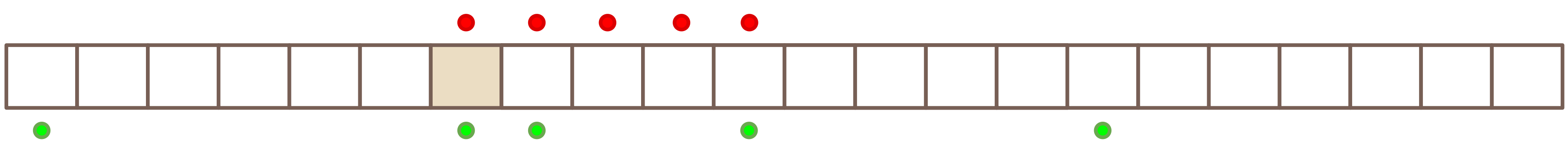
Different probing strategies

When a collision occurs, how do we search for an empty space?

linear probing:
search the array in order:
i, **i+1**, **i+2**, **i+3** . . .

quadratic probing:
search the array in nonlinear sequence:
i, **i+1**$^{\mathbf{2}}$, **i+2**$^{\mathbf{2}}$, **i+3**$^{\mathbf{2}}$ . . .

clustering:
problem where nearby hashes have very similar probe sequence so we get more collisions



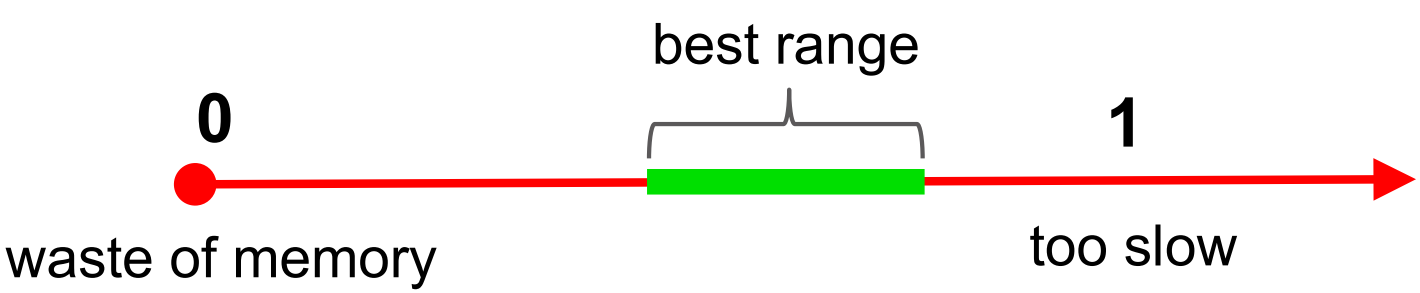
Load Factor

Load factor

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

What happens when the array becomes too full?
i.e. load factor gets a lot bigger than ½?

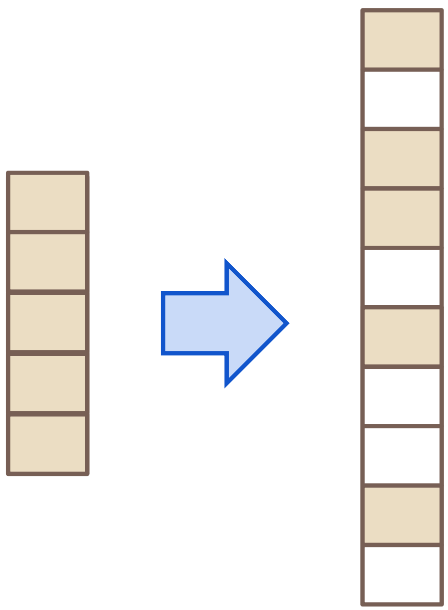
no longer expected
constant time operations



Resizing

Solution: ***Dynamic resizing***

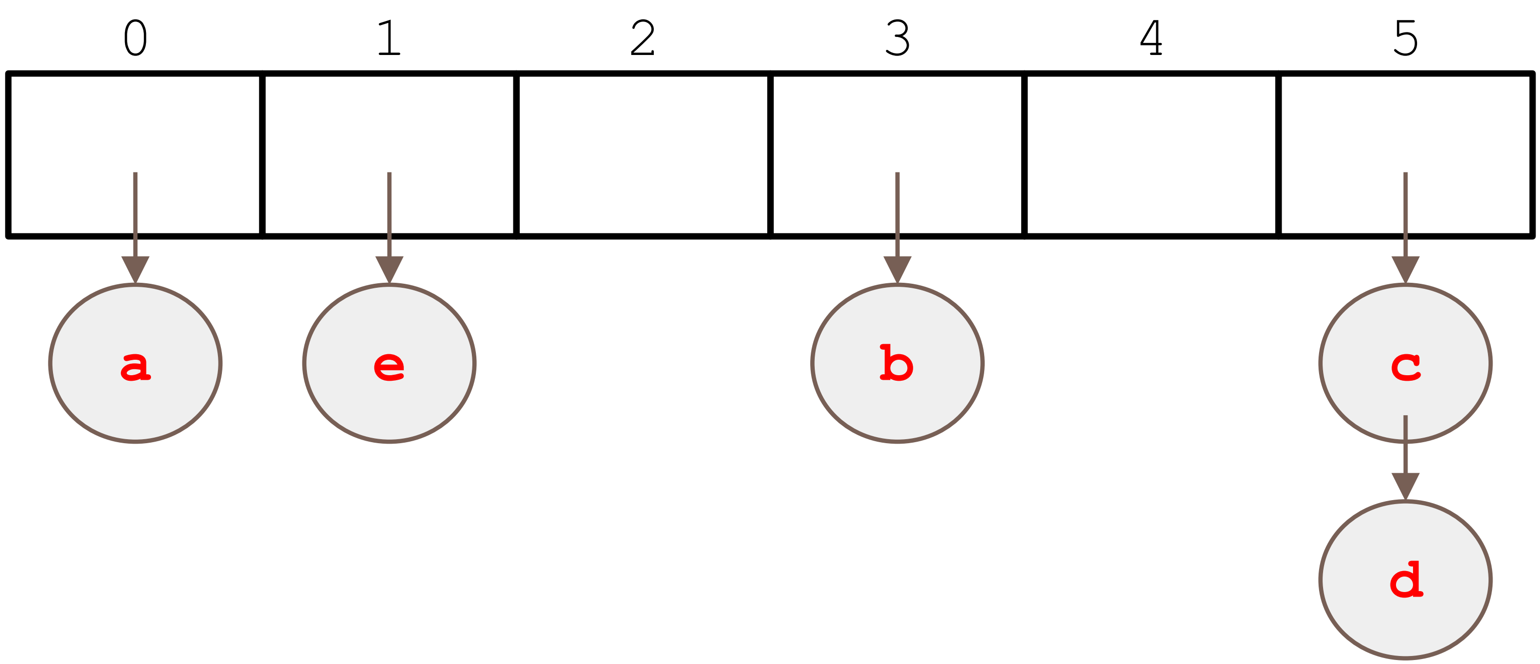
- double the size.
- reinsert / rehash all elements to new array
- Why not simply copy into first half?



Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19



Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

0	1	2	3	4	5
a	d	e	b		c

Note: Using linear probing, no resizing

Poll

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

0	1	2	3	4	5

What is the final state of the hash table if you use open addressing with quadratic probing (assume no resizing)?

Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

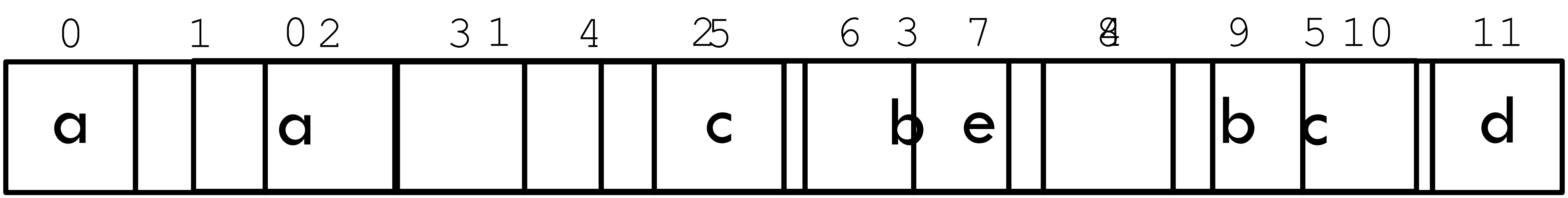
0	1	2	3	4	5
a	e	d	b		c

Note: Using quadratic probing, no resizing

Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

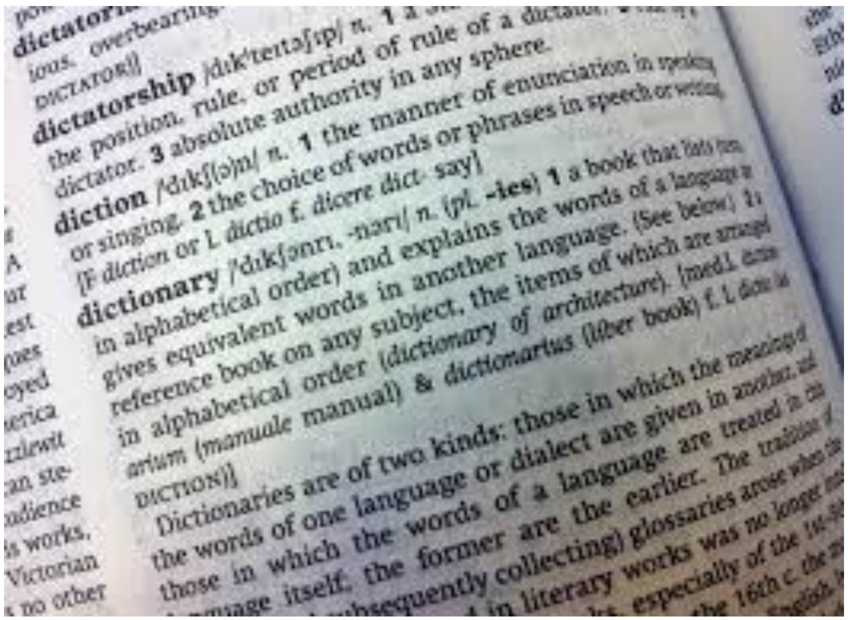


Note: Using quadratic probing, resizing if load > ½

Collision Resolution Summary

Chaining	Open Addressing
store entries in separate chains (linked lists) can have higher load factor/degrades gracefully as load factor increases	store all entries in table use linear or quadratic probing to place items uses less memory clustering can be a problem — need to be more careful with choice of hash function

Application: Hash Map

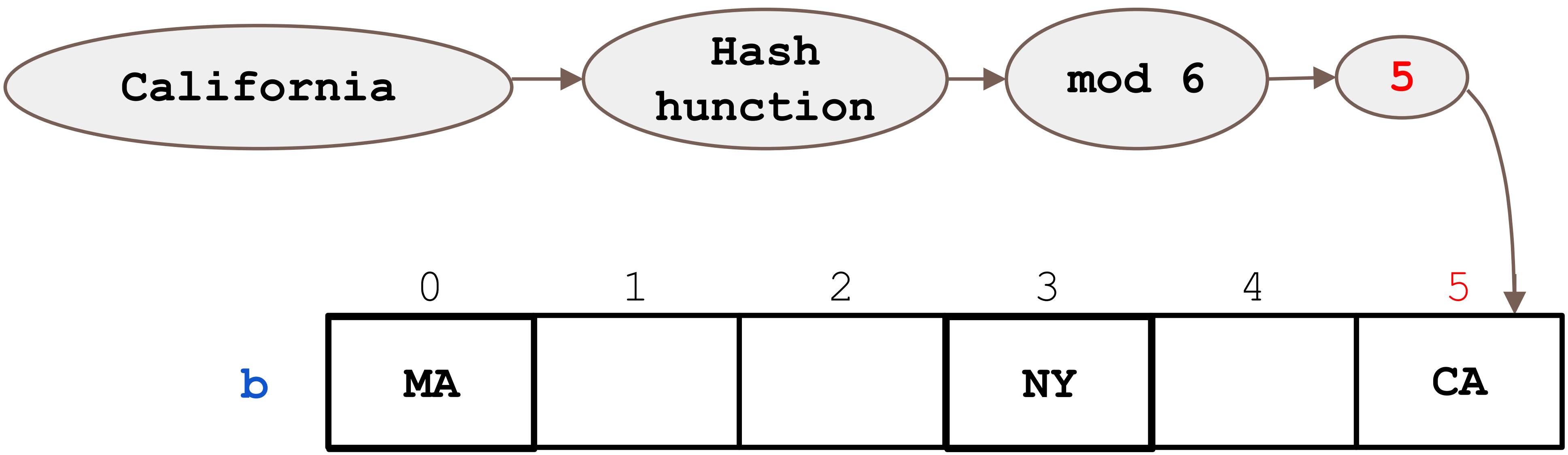


```
Map<K,V>{

    void put(K key, V value);

    void update(K key, V value);

    V get(K key);

    V remove(K key);

}
```

Application: Hash Map

Idea: finding an element in an array takes constant time when you know which index it is stored in

```
put("California","CA")
get("California")
```



HashMap in Java

- Computes hash using key.hashCode()
  - No duplicate keys
- Uses chaining to handle collisions
- Default load factor is .75
- Java 8 attempts to mitigate worst-case performance by switching to a BST-based chaining!

Hash Maps in the Real World

- Network switches
- Distributed storage
- Database indexing
- Index lookup (e.g., Dijkstra's shortest-path algorithm)
- Useful in lots of applications…