

dictatorial /ˌdɪktəˈtɔːriəl/

adv. like a dictator. 2 overbearing. [Latin: related to DICTATOR]

diction /ˈdɪksj(ə)n/ n. manner of enunciation in speaking or singing. *dictio* from *dico* dict- say]

dictionary /ˈdɪksjənəri/ n. (pl. dictionaries) a book listing (usu. alphabetically) the words of a language, explaining the words and giving corresponding words in other languages. 2 reference book explaining the terms of a particular subject.

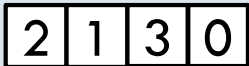

Announcements

2

- Submit Prelim 2 conflicts by **Thursday night**
- A6 is due Nov 7 (*tomorrow!*)

Ideal Data Structure

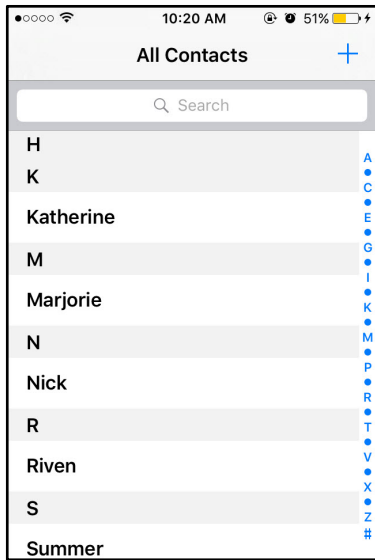
3

Data Structure	add (val x)	get (int i)	contains (val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
Goal:	$O(1)$	$O(1)$	$O(1)$

AKA add, lookup, search

Mystery Data Structure in Your Life

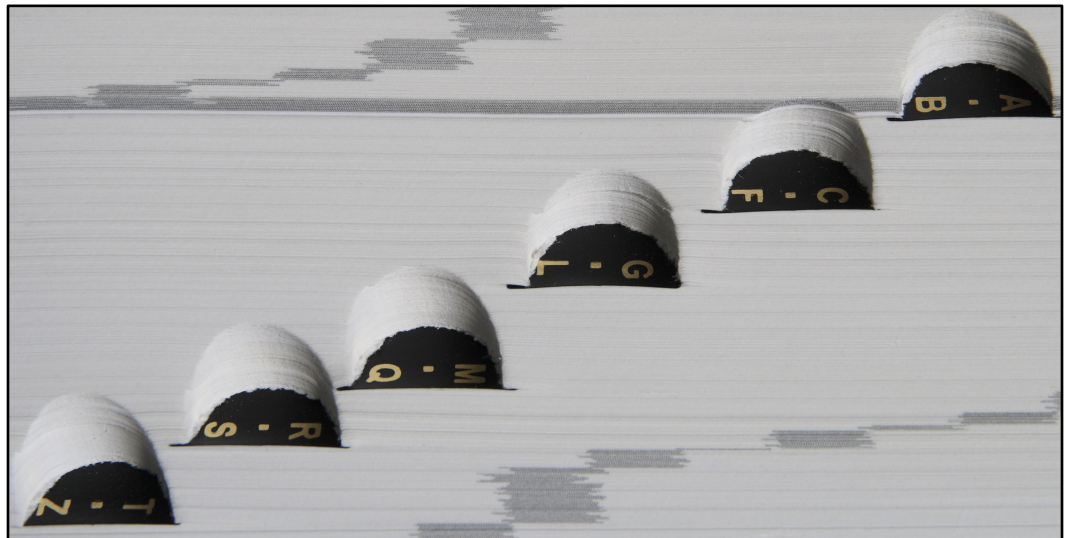
4



What do these data structures have in common?

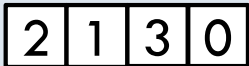

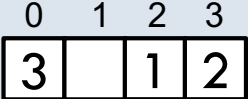


000-099 Computers, Information, & General Reference The zero hundreds is the place to look, for encyclopedias or a Bigfoot book.	100-199 Philosophy and Psychology The 100s books explain feelings you've had, from happy to grumpy or even sad.	200-299 Religion The 200s have religious books to read, or myths and legends, if that is what you need.	300-399 Social Sciences The 300s books tell of holidays, and how we celebrate in many ways.	400-499 Language On the 400s shelf, languages abound. Books about Spanish or French can be found.
500-599 Science The 500s shelf is the place to explore, science, math, or a dinosaur.	600-699 Technology The 600s are your best bets, for airplanes, cars, and friendly pets.	700-799 Arts and Recreation The 700s shelf has books of all sorts, for drawing, music, and even sports.	800-899 Literature The 800s is the place to be, for jokes, riddles, and poetry.	900-999 History and Geography In the 900s there's geography, people, places, and history.



New Data Structure: Hash Set

5

Data Structure	add (val x)	get (int i)	contains (val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
HashSet 	$O(1)$	$O(1)$	$O(1)$

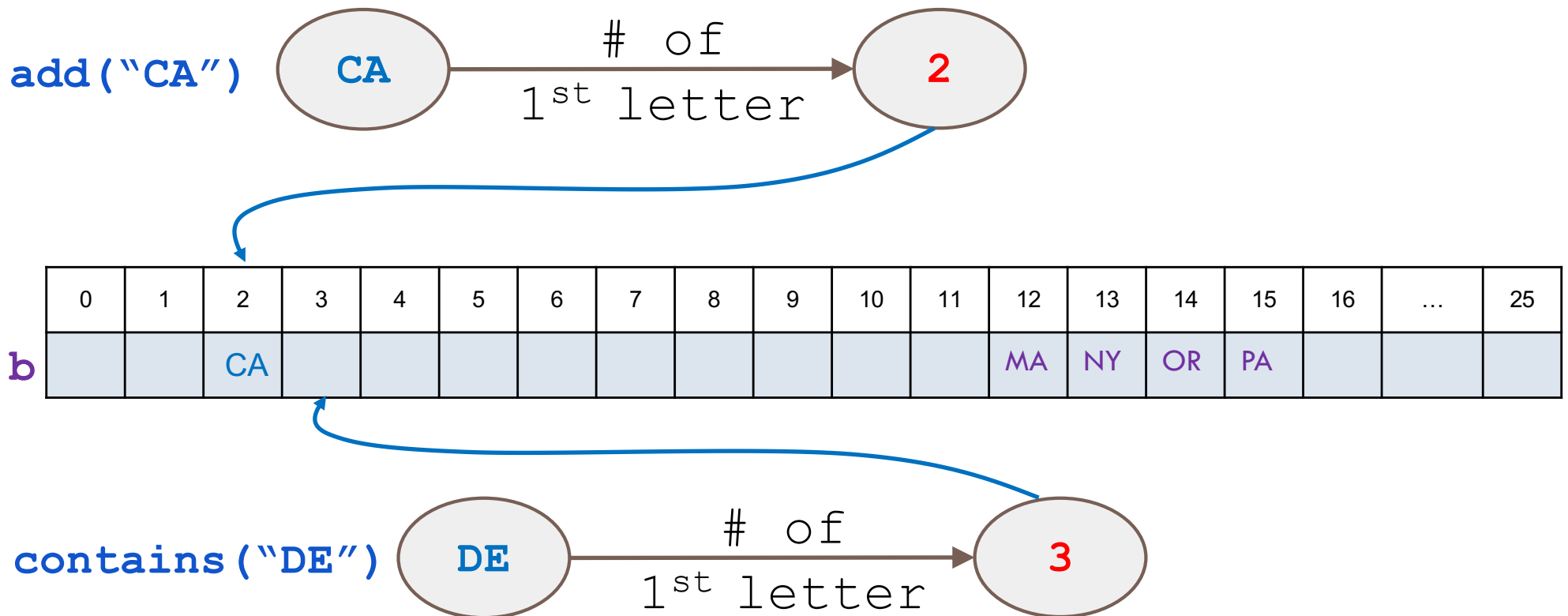
Expected time
Worst-case: $O(n)$

AKA add, lookup, search

Intuition behind a Hash Set



Idea: finding an element in an array takes constant time when you know which index it is stored in.
So... let's place elements in the array based on their starting letter! (A=0, B=1, ...)

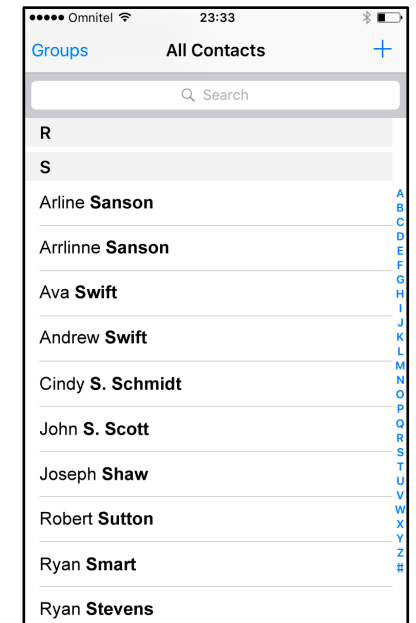
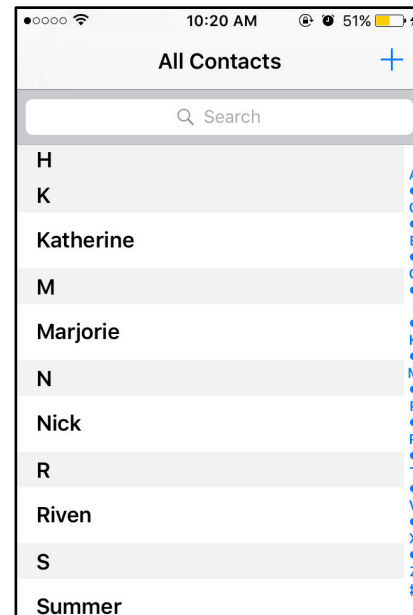


What could possibly go wrong?

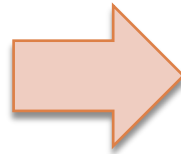
b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	25
AL		CA	DE		FL	GA						MA	NY	OR	PA			

- Some buckets get used quite a bit!
 - ▣ called **Collisions**
- Not all buckets get used



Hash Functions



1 0
4 1
3

□ Requirements:

- 1) deterministic
- 2) return a number*

□ Properties of a good hash:

- 1) fast
- 2) collision-resistant
- 3) evenly distributed
- 4) hard to invert

* the number is either in $[0..n-1]$ where n is the size of the Hash Set, or you compute the hash and then $\% n$, constraining it to be in $[0..n-1]$

Example: hashCode()

9

- Method defined in `java.lang.Object`
- Default implementation: uses memory address of object
 - ▣ If you override `equals`, you must override `hashCode`!!!
- String overrides `hashCode`:
$$s.\text{hashCode}() := s[0] * 31^{n-1} + s[1] * 31^{n-2} + \dots + s[n-1]$$

Do we like this hashCode?

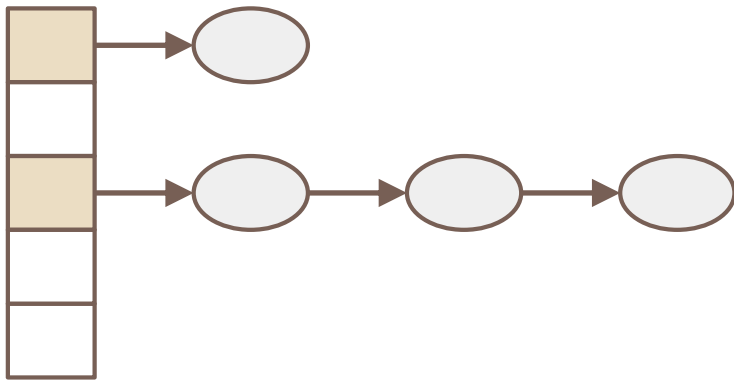
Can we have perfect hash functions?

- 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

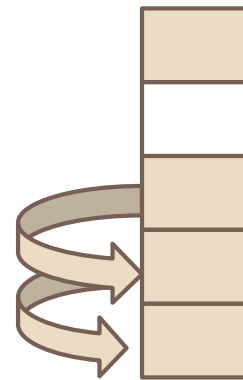
Collision Resolution

Two ways of handling collisions:

1. Chaining



2. Open Addressing

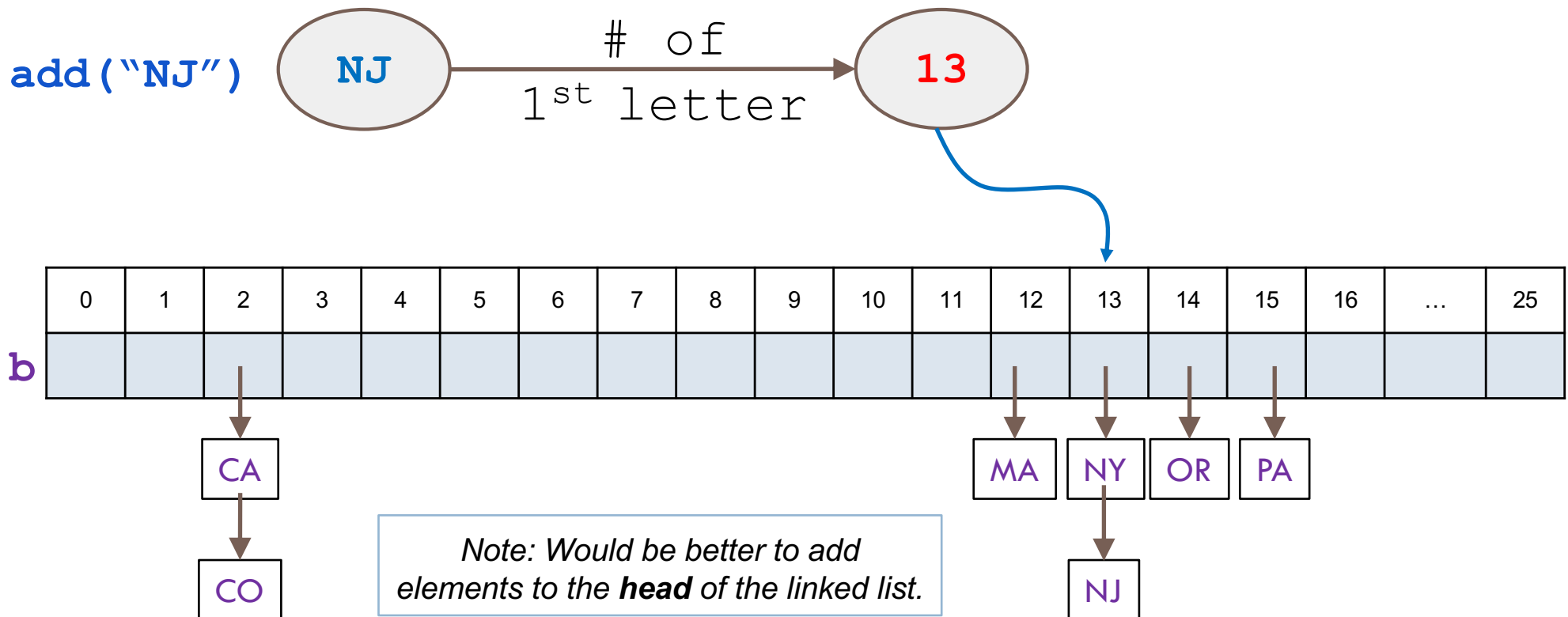


Chaining (2)

```
add("NY")
```

```
add("NJ")
```

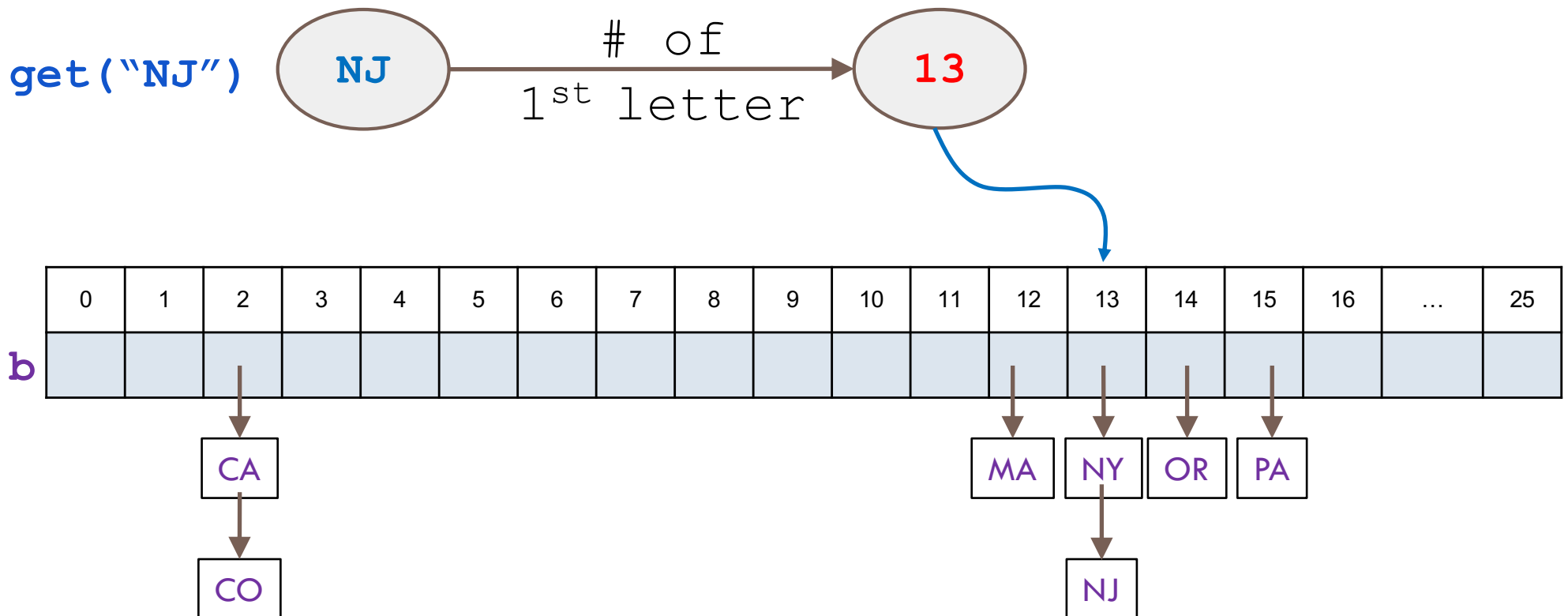
Each bucket is the beginning of a Linked List



Chaining (3)

```
add("NY")  
add("NJ")  
get("NJ")
```

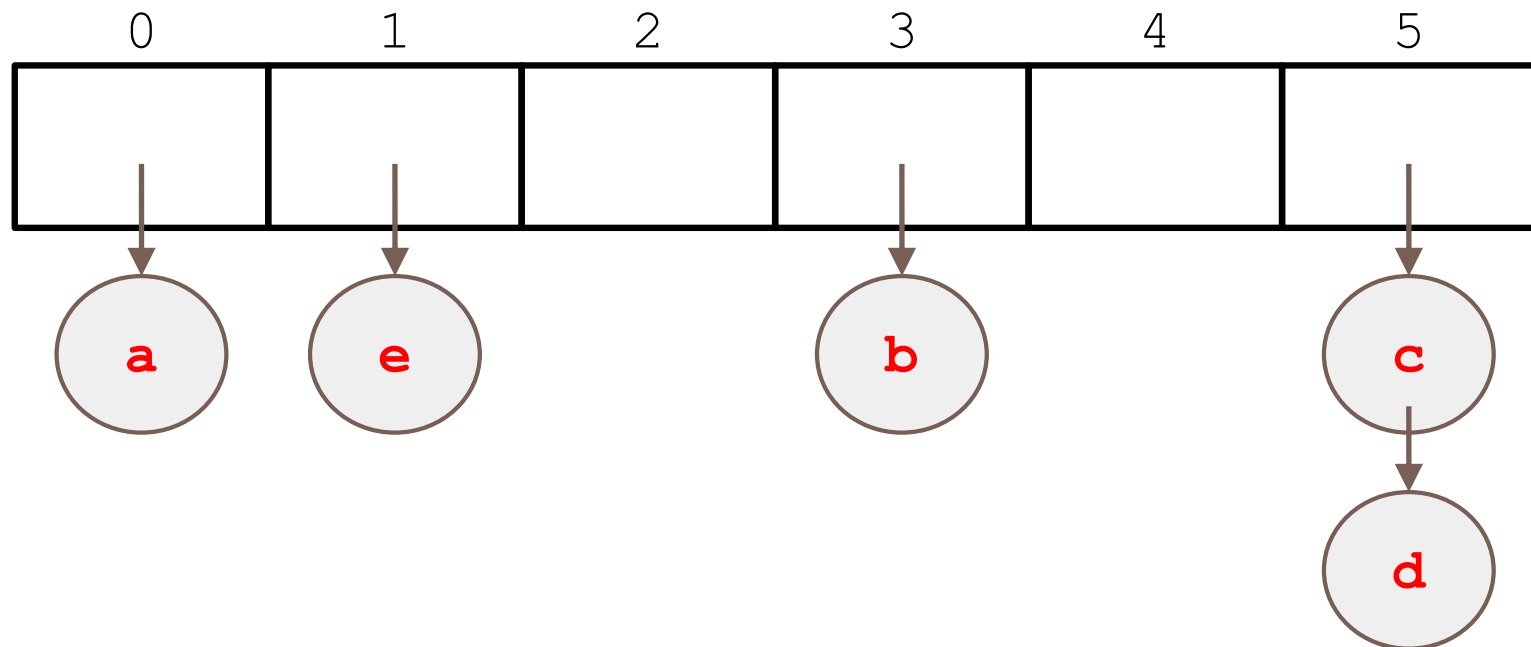
Each bucket is the beginning of a Linked List



Chaining in Action

Insert the following elements (in order) into an array of size 6:
if $\text{code} > n$, use $(\text{code} \% n_buckets)$

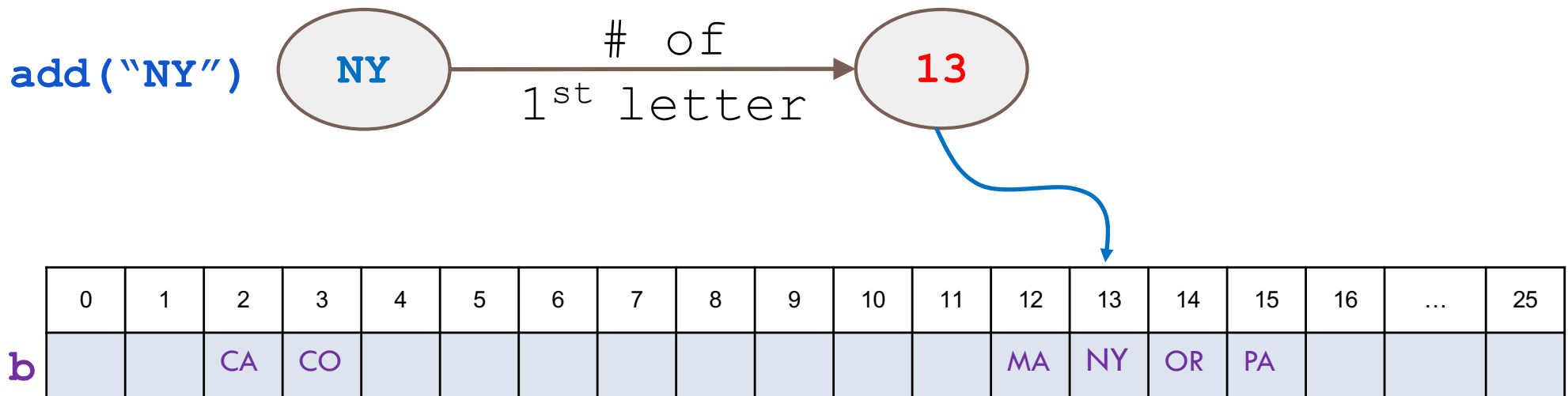
element	a	b	c	d	e
hashCode	0	9	17	11	19



Open Addressing (1)

add("NY")

Probing: Find another available space

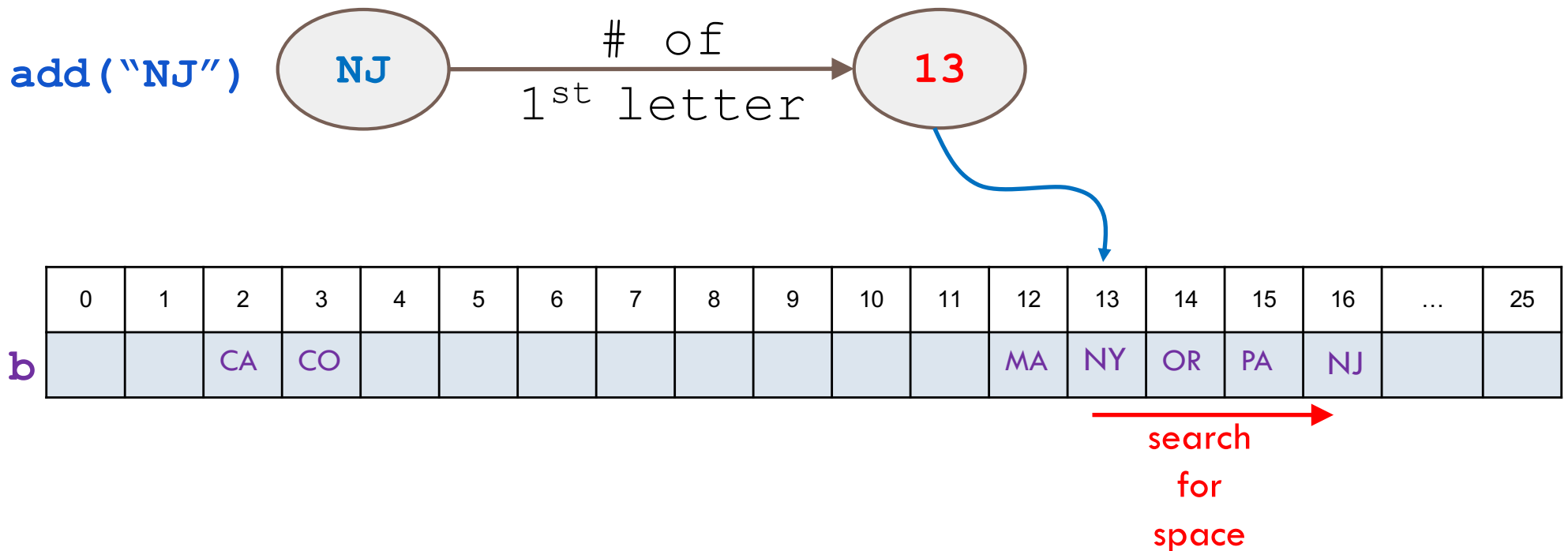


Open Addressing (2)

add("NY")

add("NJ")

Probing: Find another available space



Open Addressing (3)

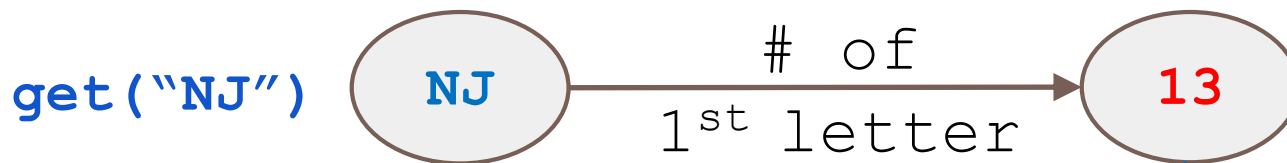
```
add("NY")
```

```
add("NJ")
```

```
...
```

```
get("NJ")
```

Probing: Find another available space



b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	25
		CA	CO									MA	NY	OR	PA	NJ		

Search for NJ
(stop searching if
element is NULL)

What could possibly go wrong?

```
add("NY"), add("NJ"), get("NY"), get("NJ")
```

Deletion Problem w/Open Addressing

Probing: Find another available space

```
add ("NY")
```

```
add ("NJ")
```

```
get ("NY")
```

```
get ("NJ")
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	25
b			CA	CO									MA	NY	OR	PA	NJ		



Search for NJ
(stops searching b/c
element b[13] is NULL!)

Deletion Solution for Open Addressing

Probing: Find another available space

Need to mark element as “not present”

Indicates to search that it should keep looking

```
add ("NY")
```

```
add ("NJ")
```

```
get ("NY")
```

```
get ("NJ")
```

b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	25
		CA	CO									MA	NY NP	OR	PA	NJ		



Search for NJ

(searches until it finds a NULL element or the present element it's looking for)

Different probing strategies

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

clustering:

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

linear probing:

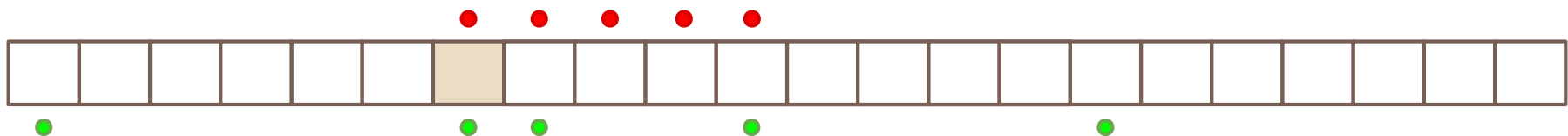
search the array in order:

$i, i+1, i+2, i+3 \dots$

quadratic probing:

search the array in nonlinear sequence:

$i, i+1^2, i+2^2, i+3^2 \dots$



In order to have access to every bucket, important for size to be a prime number when using quadratic probing.

Linear Probing in Action

Insert the following elements (in order) into an array of size 5:

element	a	b	c	d
hashCode	0	8	17	12

0	1	2	3	4
a		c	b	d

probe #1
inserting d:
i
full!

probe #2
inserting d:
i+1
full!

probe #3
inserting d:
i+2
has
space!

Quadratic Probing in Action

Insert the following elements (in order) into an array of size 5:

element	a	b	c	d
hashCode	0	8	17	12

0	1	2	3	4
a	d	c	b	

probe #3
inserting d:
 $i+2^2$
has
space!

probe #1
inserting d:
 i
full!

probe #2
inserting d:
 $i+1^2$
full!

Load Factor

24

Load factor

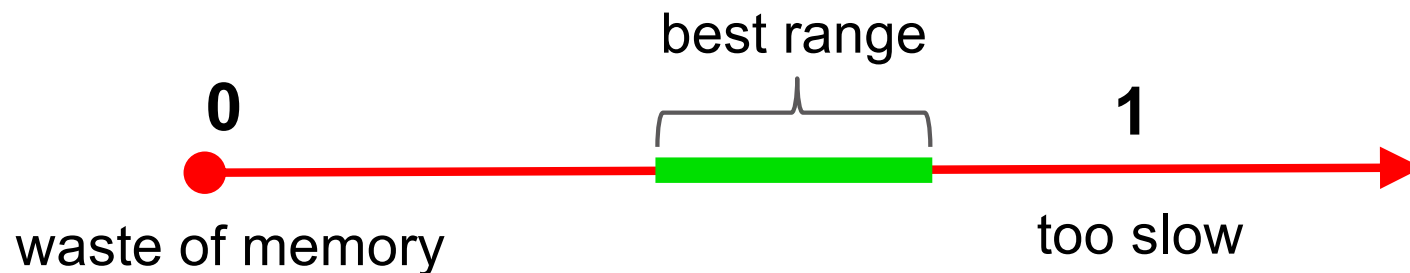
$$\lambda = \frac{\text{\# of entries}}{\text{length of array}}$$

If load factor = $\frac{1}{2}$, expected # of probes is 2.

What happens when the array becomes too full?

i.e. load factor gets a lot bigger than $\frac{1}{2}$?

**no longer expected
constant time operations**

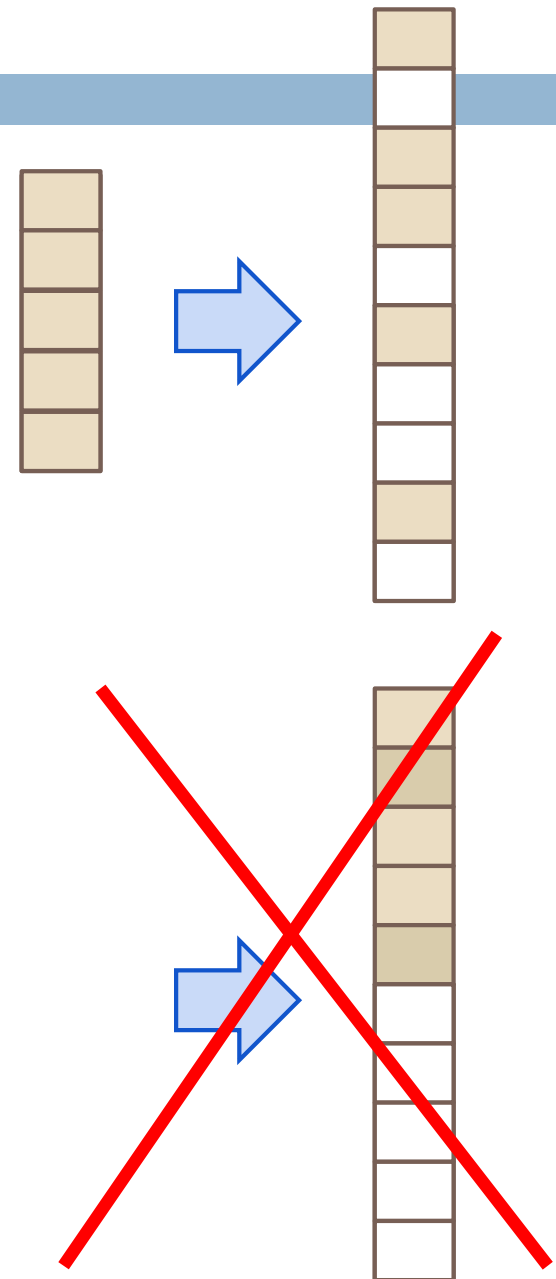


Resizing

Solution: ***Dynamic resizing***

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

*if using quadratic probing, use a prime $>2n$



Collision Resolution Summary

26

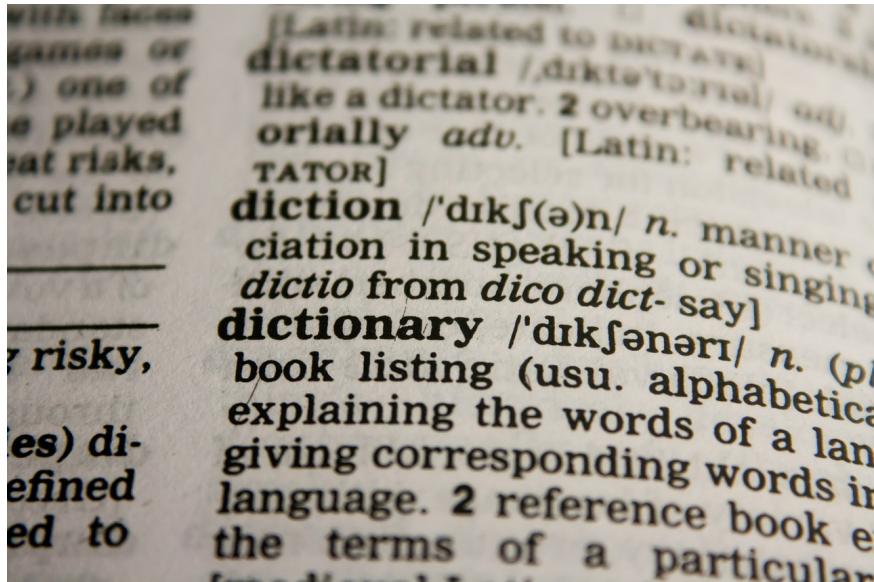
Chaining

- store entries in separate chains (linked lists)
- can have higher load factor/degrades gracefully as load factor increases

Open Addressing

- 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



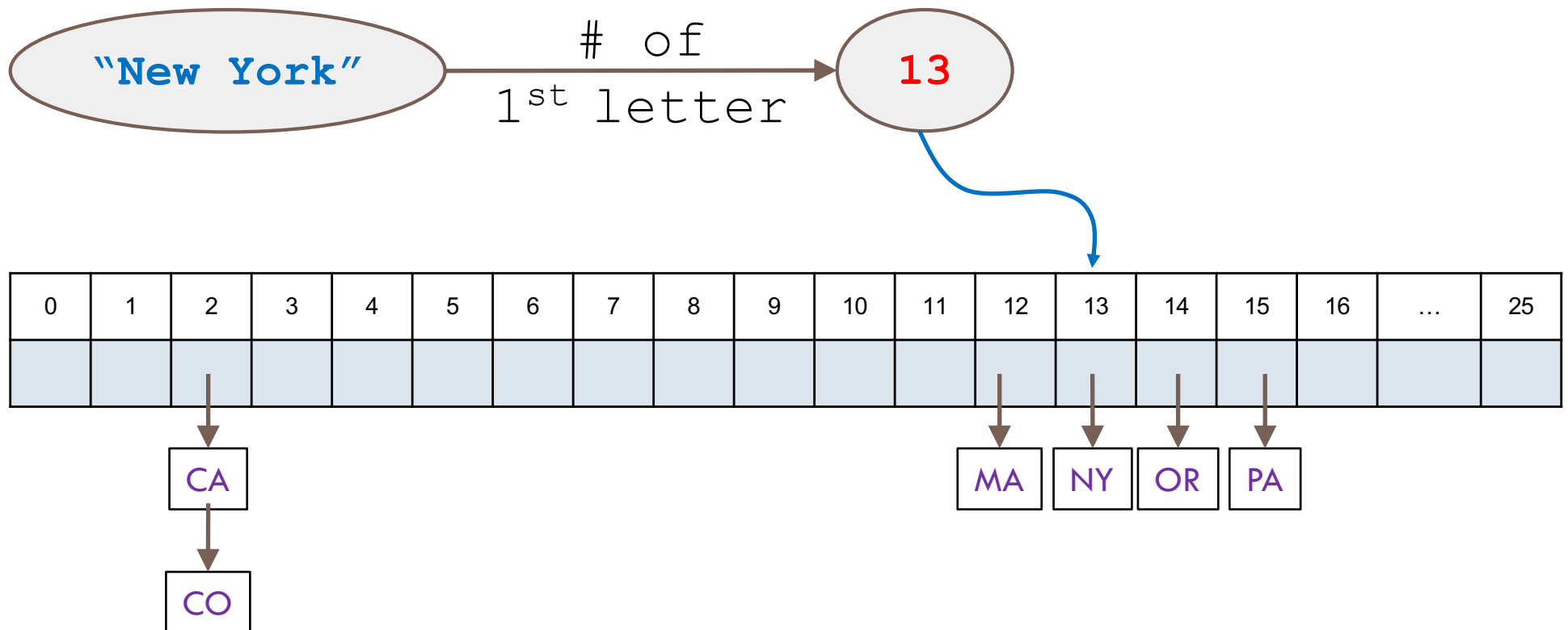
- Use the **key** for lookups
- Store the **value**

Example: **key** is the word, **value** is its definition

```
Map<K, V>{  
  
    void put(K key, V value);  
  
    void update(K key, V value);  
  
    V get(K key);  
  
    V remove(K key);  
  
}
```

Hash Map (1)

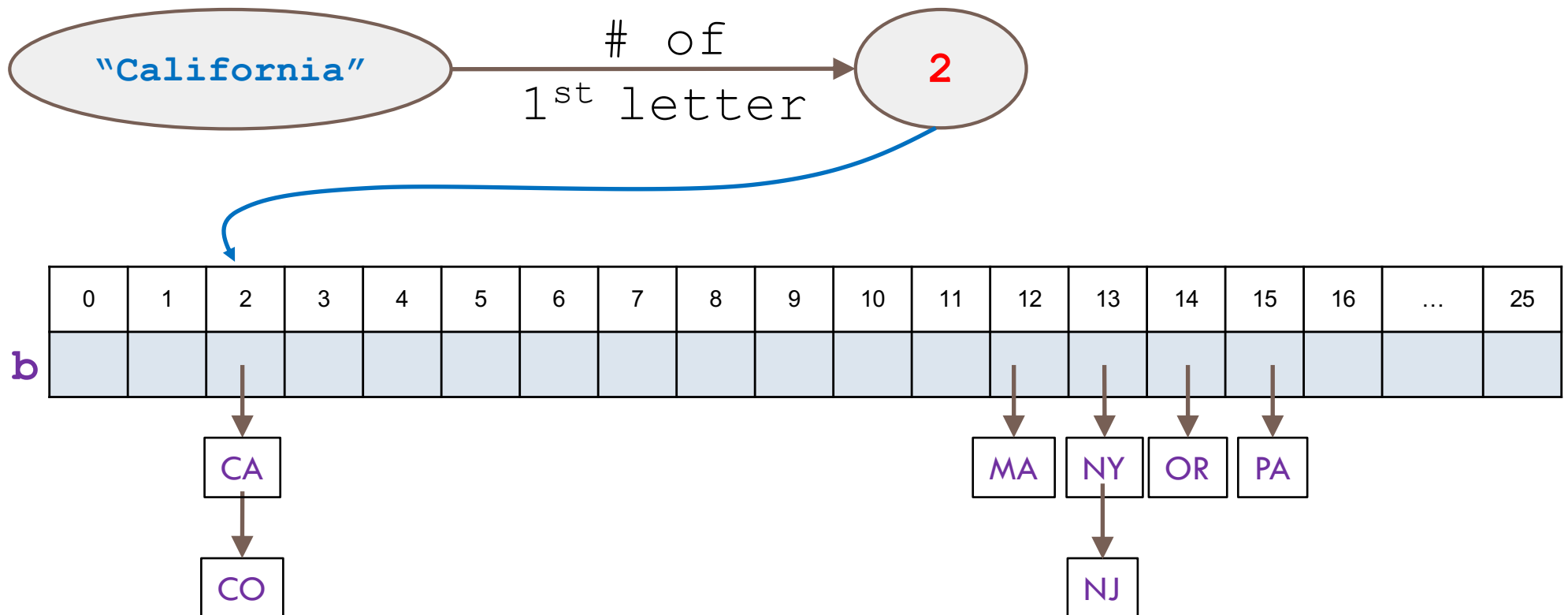
```
put("New York", "NY")
```



Hash Map (2)

```
put("New York", "NY")
```

```
get("California")
```



HashMap in Java

30

- 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

31

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