

dictatorial /ˈdɪktəˈtɔːriəl/

like a dictator. 2 overbearing.
dictatorially *adv.* [Latin: related to TATOR]

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

dictionary /ˈdɪksənəri/ *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

Announcements

2

- Submit Prelim 2 conflicts by **Wednesday (tomorrow) night**
- A6 is due April 18 (*Thursday!*)
- Prof Clarkson diagnosed with a concussion and is staying home this week. Don't send him email—he's supposed to stay away from his computer.

Material in for Hashing

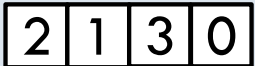

3

- Tutorial on hashing:
in lower navigation bar in JavaHyperText
- Entry `hash` in JavaHyperText
- Specific to Java. API documentation for:
`hashCode()` and function `equals(Object ob)`
- Lecture notes page of course website.
Demo code for hashing with chaining and hashing
with open addressing

Ideal Data Structure

4

Table gives expected times, not worst-case times

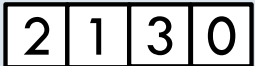

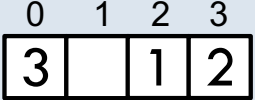
Data Structure	add (val x)	get (int i)	contains (val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
Goal:	$O(1)$	$O(1)$	$O(1)$

Also known as: add, lookup, search

New Data Structure : Hash Set

5

Table gives expected times, not worst-case times

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

Notion of hashing

6

Hash: to chop to pieces; to make a confused muddle of; to jumble; to dice, chop, mince.

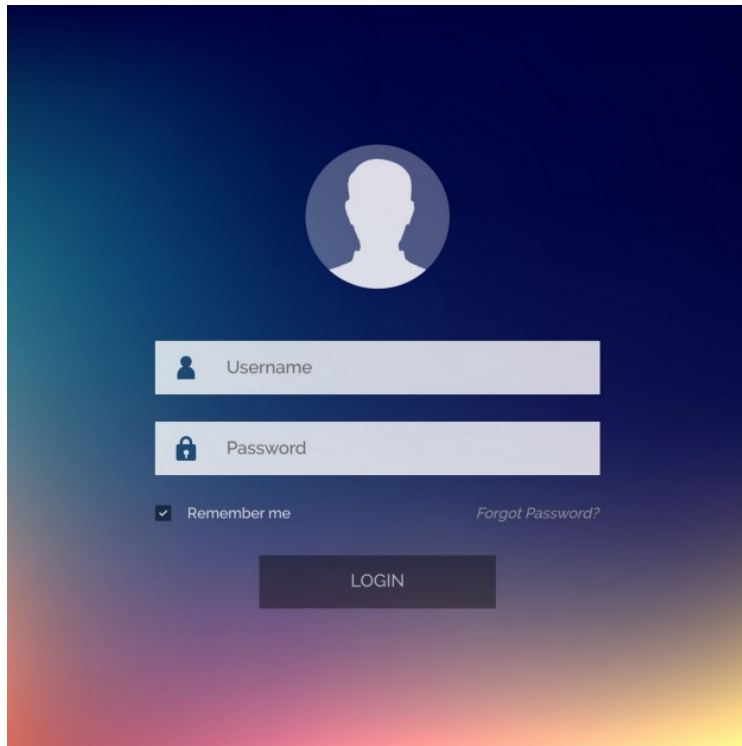
In computing: Produce a relatively small number or string from something a lot bigger, like a file, or an Java object.

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

Look at CMS page for A2 submission. Md5 is a **hash function**. Given your A2.java file, it produces a 128-bit number from it. Sometimes called a **checksum**. Compare the Md5 number for your for your file to the MD5 number of the one that was uploaded. If different, uploading corrupted the file.

Application: Password Storage

- Hash functions are used to store passwords
- Could store plaintext passwords
 - ▣ Problem: Password files get stolen



$h(\text{password})$: h is the hash function.
It produces some jumbled version of the password.

Hashing history



We will use hashing—a hash function—to implement sets of values in a **hash table**.

1953. Hand Peter Luhn wrote an internal IBM memorandum that used **hashing with chaining**.

A few others did it roughly the same time.

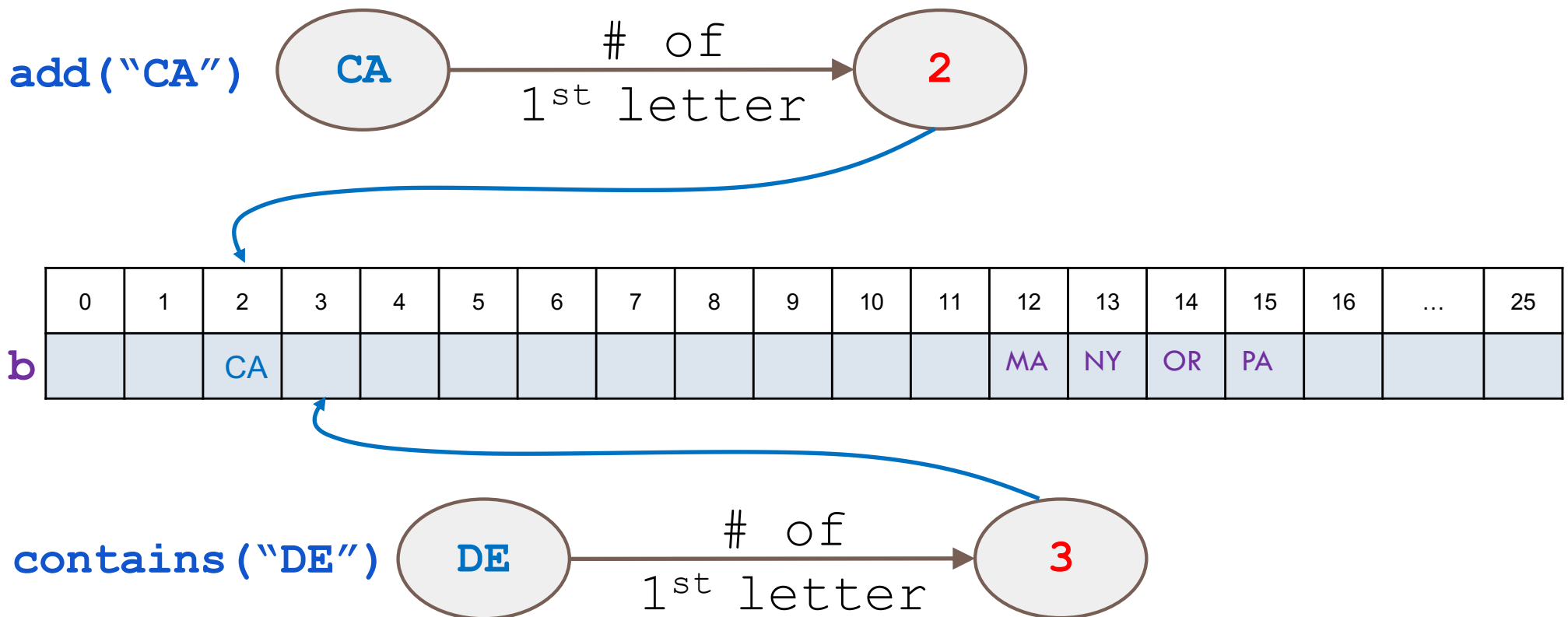
Ershov (Russian) and Amdahl independently invented **hashing with open addressing and linear probing**.

Intuition behind a Hash Set



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

So... let's place elements in the array based on their starting letter! (A=0, B=1, ...)



What could 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			

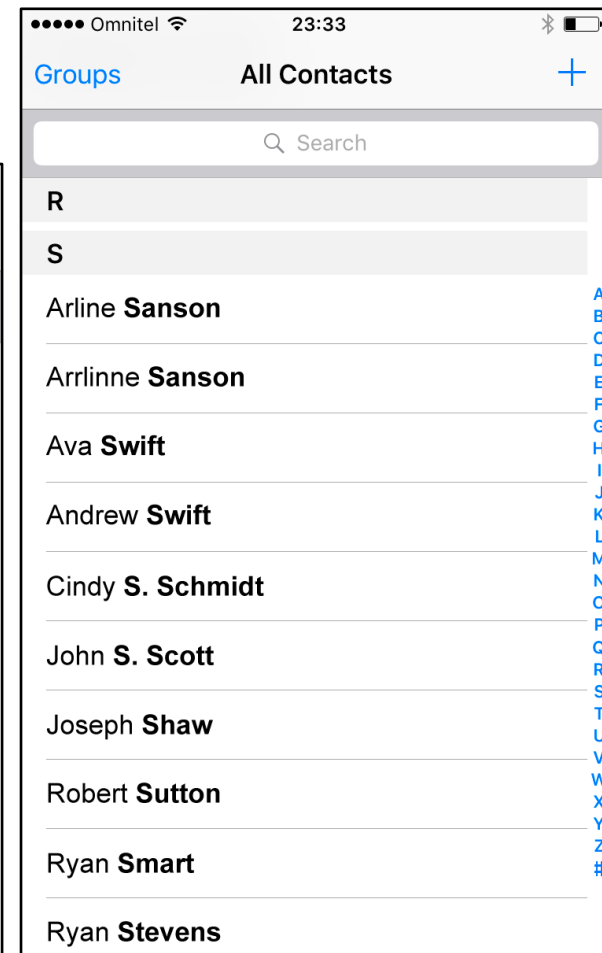
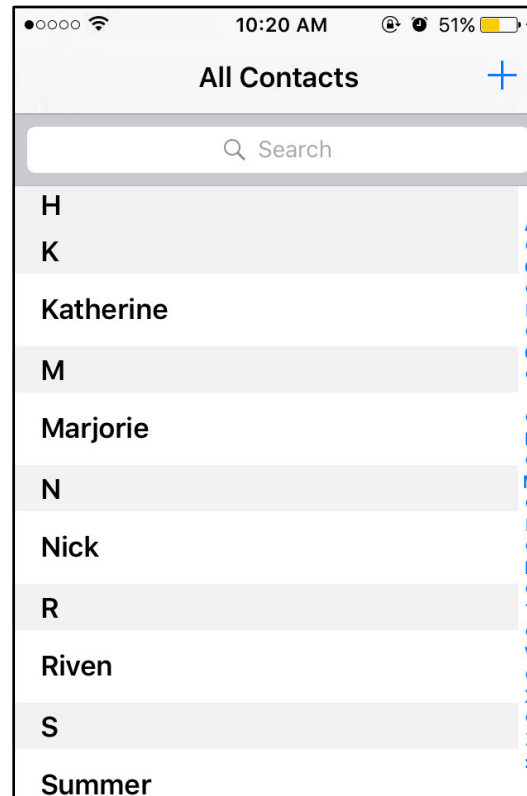
- Some buckets get used quite a bit!

Connecticut, Colorado

- called **Collisions**

- Not all buckets get used

bucket: one of the array elements



Hash Function

0	1	2	3	4	5	6	7	8	9
AL		CA	DE		FL	GA			

Given a value to be put into the table, a **hash function** returns an index where to put it.

E.g. `hash function(stateName)` could return value depending on first character:

0 for **A**, 1 for **B**, 2 for **C**, etc.

The hash function knows nothing about the table size.

Therefore, **always** take the hash-function values mod the table size in order to get an index into the table.

Example: `hashCode("Oregon") mod 10 = 14 mod 10 = 4`

So put "Oregon" in bucket 4.

Example: hashCode()

12

- hashCode() defined in java.lang.Object
- Default implementation: uses memory address of object
 - ▣ If you override equals, you must override hashCode!!!
We'll explain why later.
- String overrides hashCode:
 s.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?

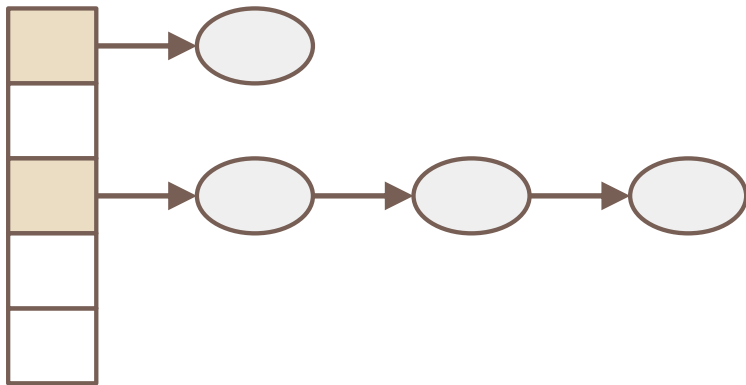
- A perfect hash function will map each value to a different index in the hash table
- Impossible in practice
 - Don't know size of the array
 - Number of possible values far far exceeds the array size
 - No point in a perfect hash function if it takes too much time to compute

Forget about perfect hash functions!

Collision Resolution

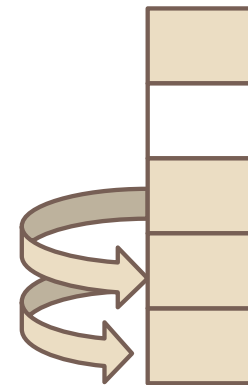
Two ways of handling collisions:

1. Chaining



A bucket contains a linked list of items that hash to it

2. Open Addressing

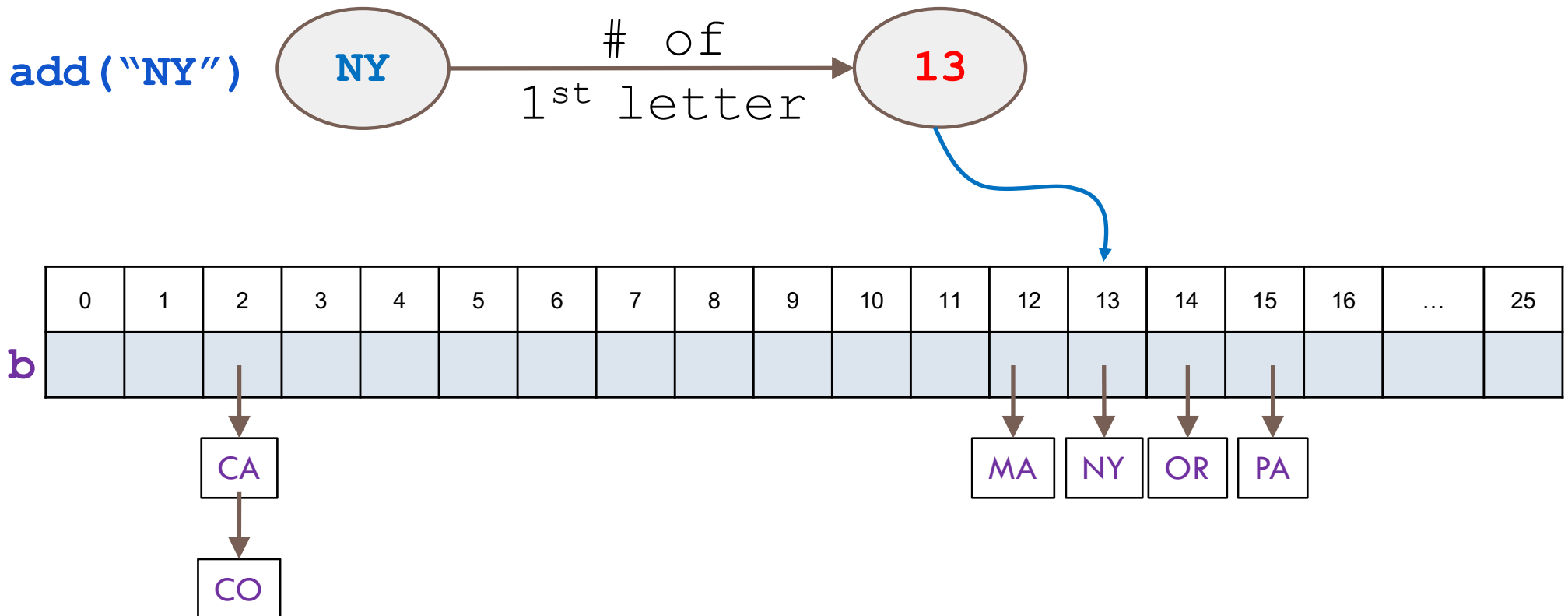


A bucket contains one item of the set. Look in successive array elements to find a place for a new item

Chaining (1)

add ("NY")

Each bucket is the beginning of a Linked List

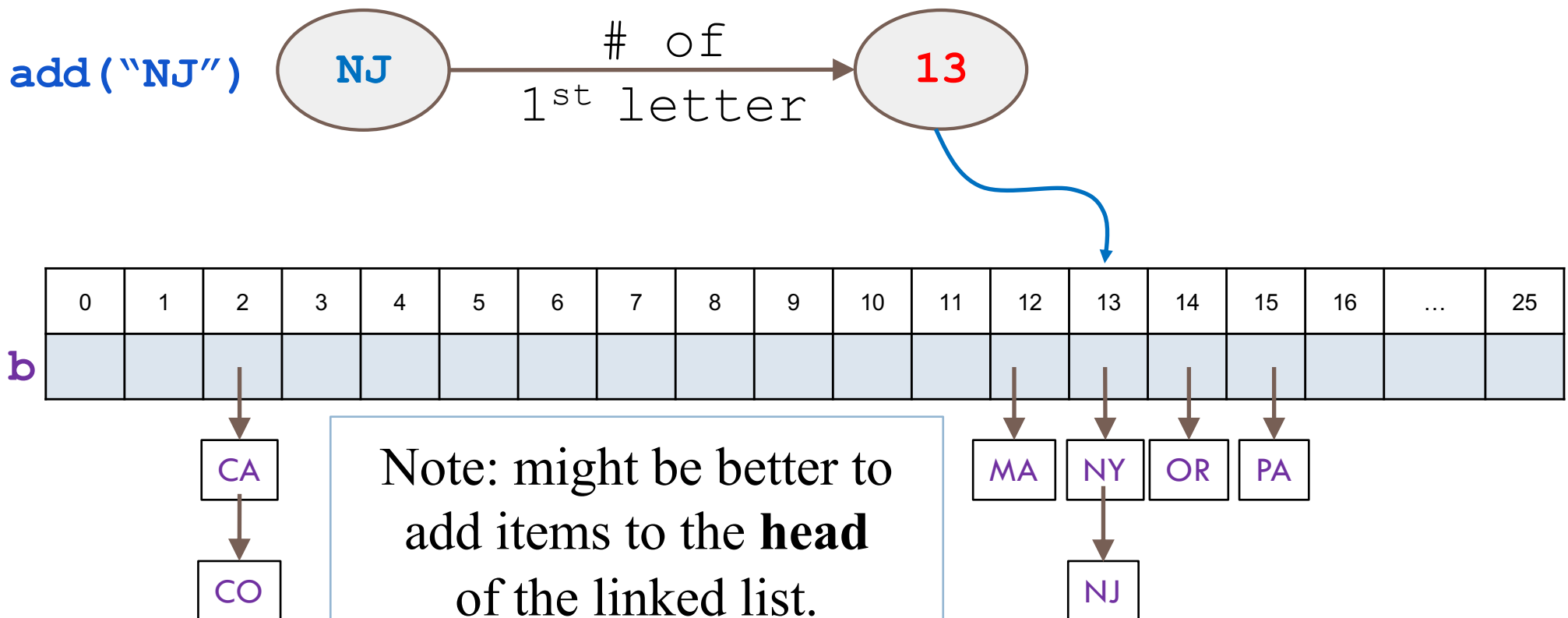


Chaining (2)

add ("NY")

add ("NJ")

Each bucket is the beginning of a LinkedList



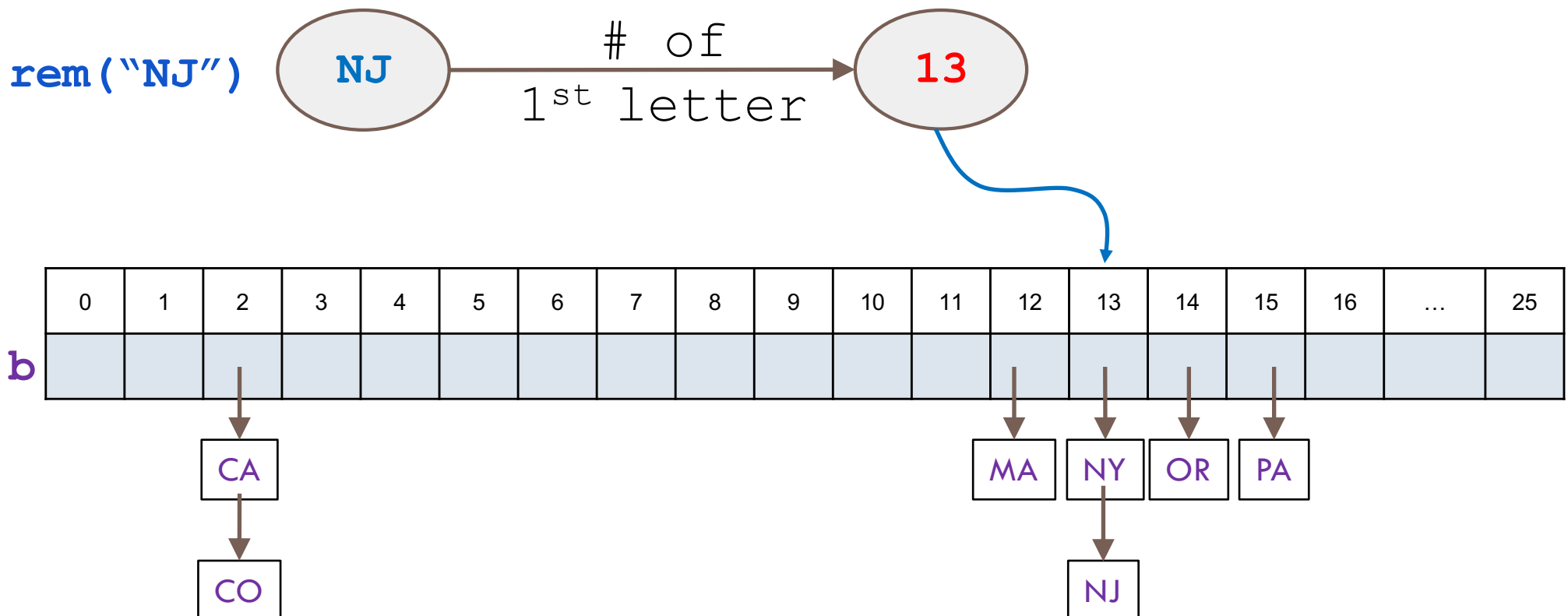
Chaining (3)

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

```
rem ("NJ")
```

Each bucket is the beginning of a LinkedList

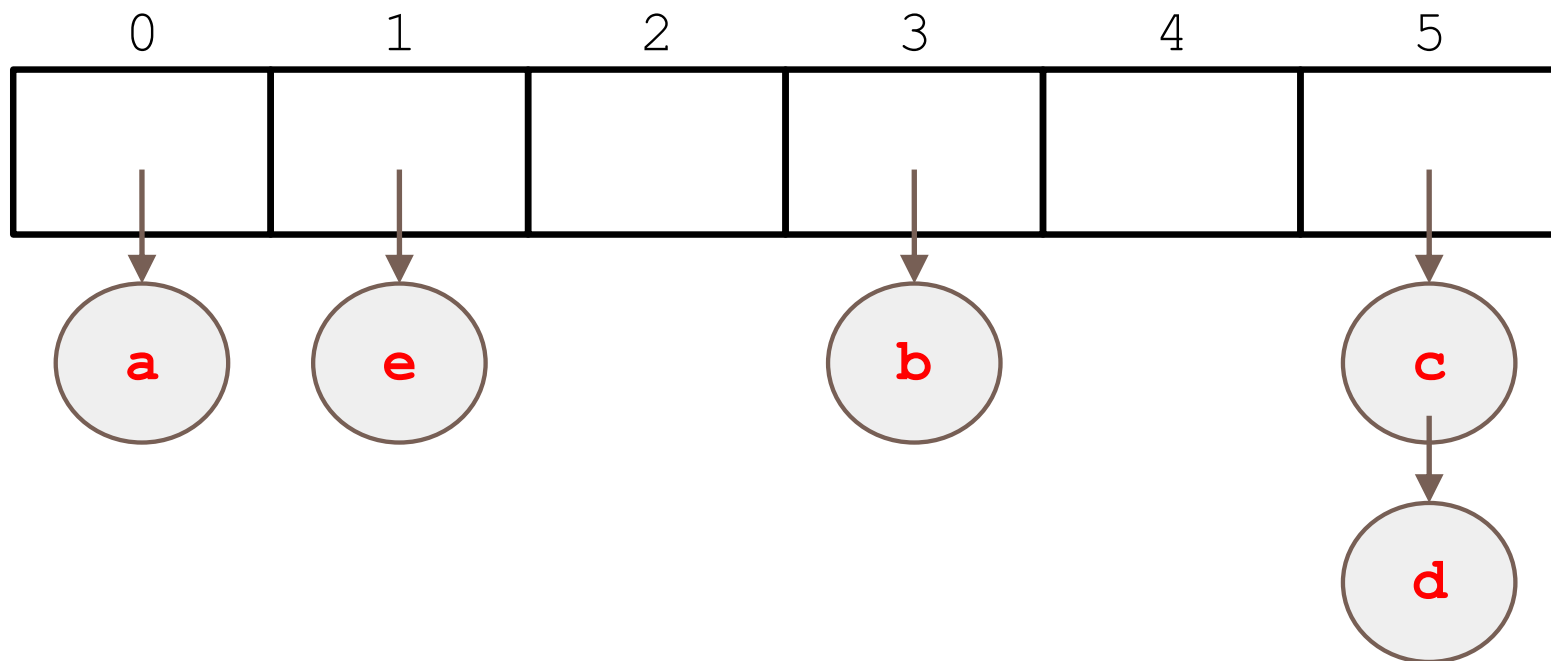
Rem: remove and return



Chaining in Action

Insert the following elements (in order) into an array of size 6:
Use $(\text{hashCode} \% \text{n_buckets})$

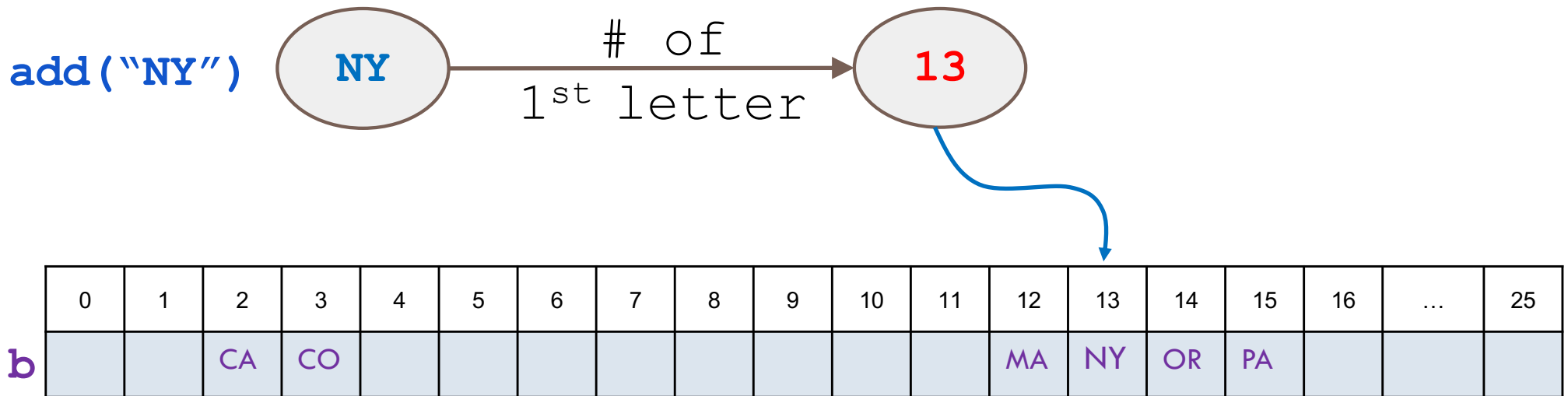
element	a	b	c	d	e
hashCode	0	9	17	11	19



Open Addressing (1)

add("NY")

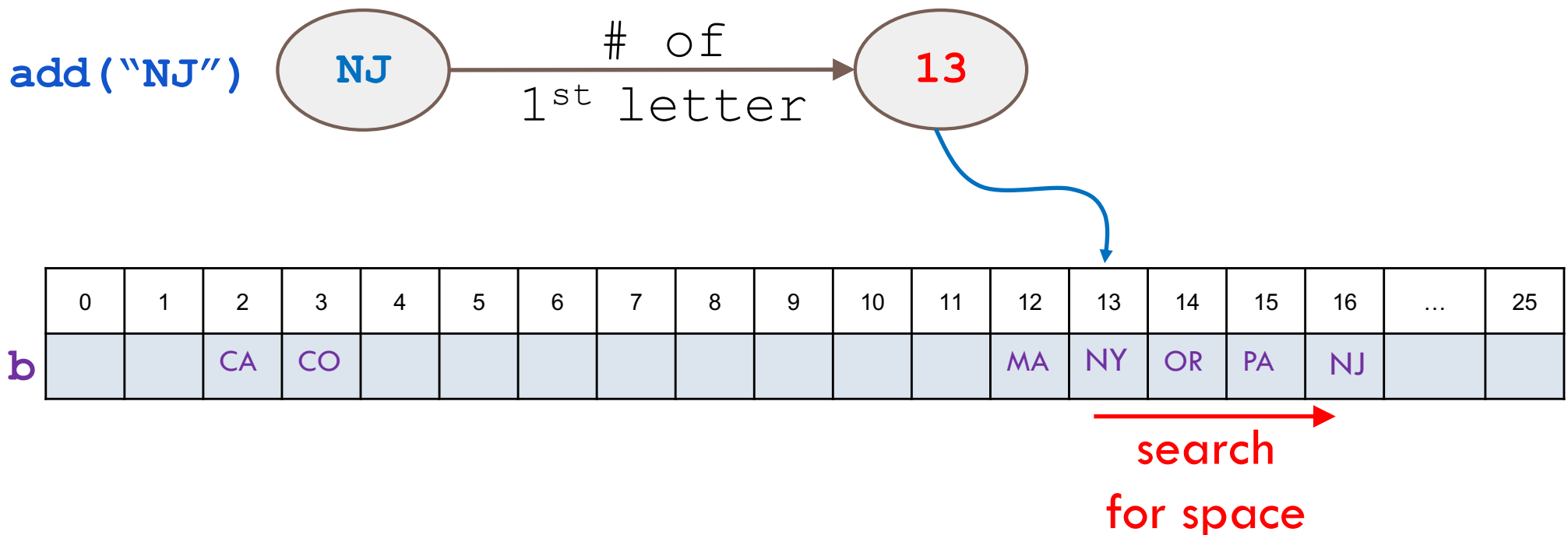
Probe: One test in finding space for a new item or when searching for an item



Open Addressing (2)

add ("NY")
add ("NJ")

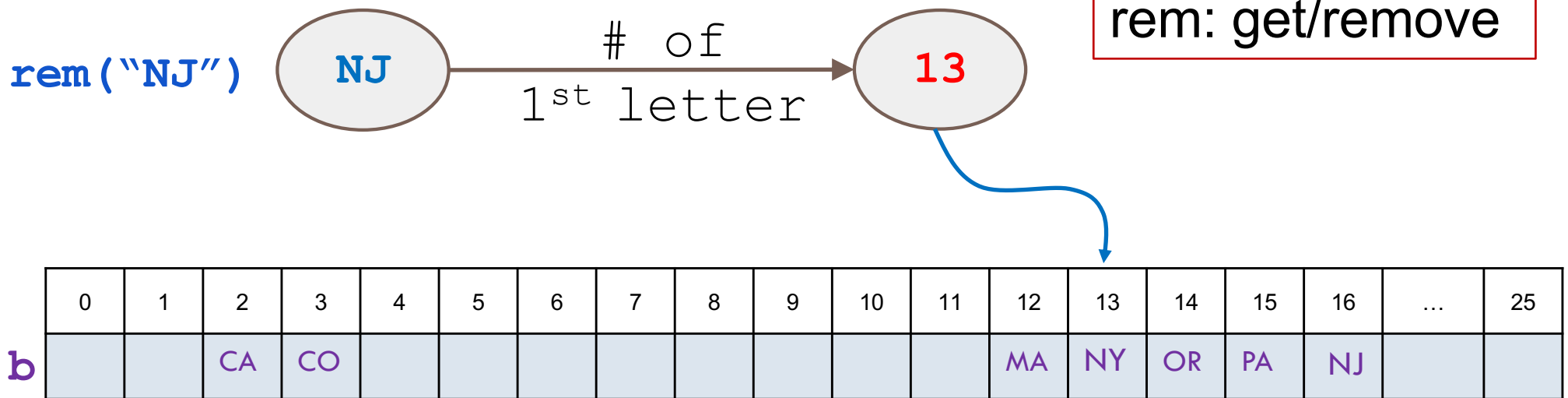
Probe: One test in finding space for a new item or when searching for an item



Open Addressing (3)

Probe: One test in finding space for a new item or when searching for an item

```
add("NY")
add("NJ")
...
rem("NJ")
```



What could possibly go wrong?

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

Search for NJ
(stop searching if
element is **null**)

Deletion Problem w/Open Addressing

Probe: One test in finding space for a new item or when searching for an item

add ("NY")

add ("NJ")

rem ("NY")

rem ("NJ")

rem: get/remove

	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

(stop searching b/c
element b[13] is **null**!)

Deletion Solution for Open Addressing

Probe: One test in finding space for a new item or when searching for an item

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	OR	PA	NJ		

NP

Search for NJ

(search until it finds a **null** element
or the element it's searching for)

Different probing strategies

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

clustering:

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

linear probing:

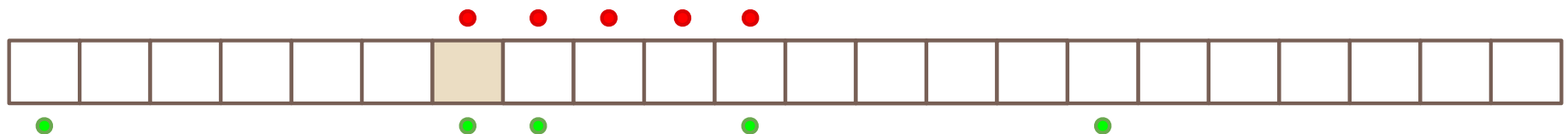
search the array
in order:

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

quadratic probing:

search the array in
this sequence:

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



Quadratic probing requires the size of the array to be a
prime in order to have access to every bucket.

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 *probe #2* *probe #3*
insert d: *insert d:* *insert d:*
i *i+1* *i+2*
full! *full!* *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 *probe #1* *probe #2*
insert d: *insert d:* *insert d:*
 $i+2^2$ *i* *$i+1^2$*
has space! *full!* *full!*

In Java, functions hashCode and equals



HashSet, HashMap use functions hashCode(), equals(...)

c.hashCode() in class Object returns the address in memory of object c

c.equals(c1) in class Object is true iff c and c1 point to the same object

In Java, functions hashCode and equals

Elements of set HashSet have class type, e.g. **Pt**

Rewrite equals

```
/** Return true iff this and ob are of the same  
 * class type, their x fields are equal, and  
 * their y fields are equal. */
```

```
public boolean equals(Object ob) {...}
```

```
Class Pt {  
    int x;  
    int y;  
    ...  
}
```

Because b and c are equal, only one of them should be put in the set

b and c are
different **Pt**
objects but
b.x = c.x
b.y = c.y

0	1	2	3	4
c			b	

In Java, functions hashCode and equals

What we learn from this

Function `hashCode` has to be defined so that:

if `b.equals(c)` is true,
then `b.hashCode() == c.hashCode()`

so that `b` and `c` hash to the same index.
The test for equality of `c` and `b` will show it's
already in.

```
Class Pt {  
    int x;  
    int y;  
    ...  
}
```

0	1	2	3	4
			c b	

In Java, functions hashCode and equals

Elements of set HashSet have class type, e.g. **Pt**

Rewrite equals

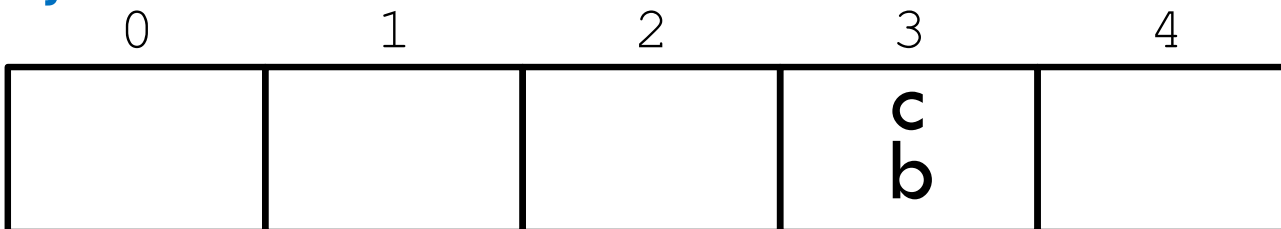
```
/** Return true iff this and ob are of the same  
 * class type, their x fields are equal, and  
 * their y fields are equal. */
```

```
public boolean equals(Object ob) {...}
```

```
public int hashCode() {  
    return abs(x + y);  
}
```

```
Class Pt {  
    int x;  
    int y;  
    ...  
}
```

b and **c** are
different **Pt**
objects but
b.x = c.x
b.y = c.y



Load Factor

31

Load factor

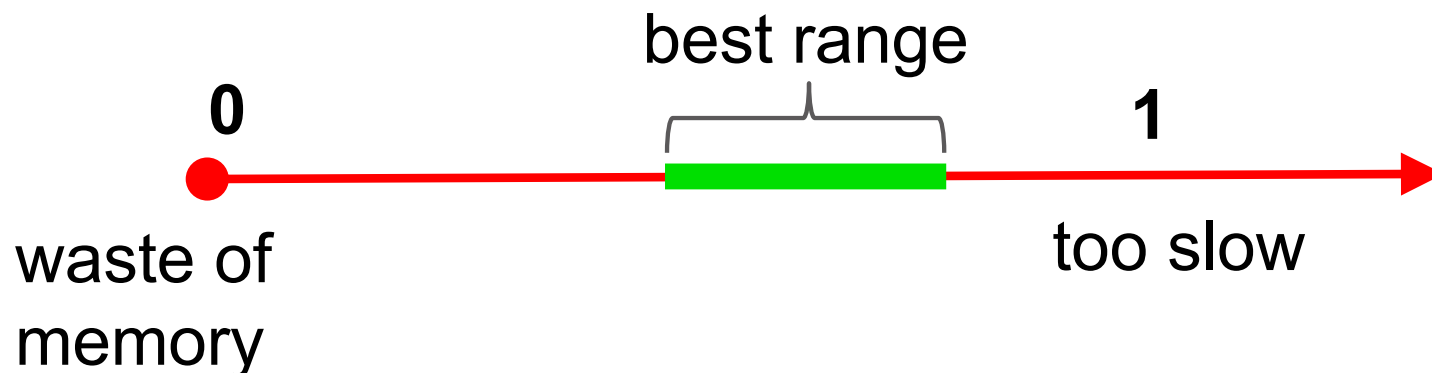
$$\lambda = \frac{\text{\# of entries}}{\text{length of array}}$$

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**



Chaining: Worst case time $O(n)$

32

	0	8999
Chaining worst case time	8999 nulls, 1 list of size 6000	

Suppose everything hashes to the last array element, so that all array elements are null except the last, and that last linked list has n elements in it ---the set has size n .

In this case, operations **add**, **contains**, and **remove** all take time $O(n)$. That's the worst case.



Linear probing: Worst case time $O(n)$

33

Chaining worst case time

0	n	8999
b	n elements	null null ... null

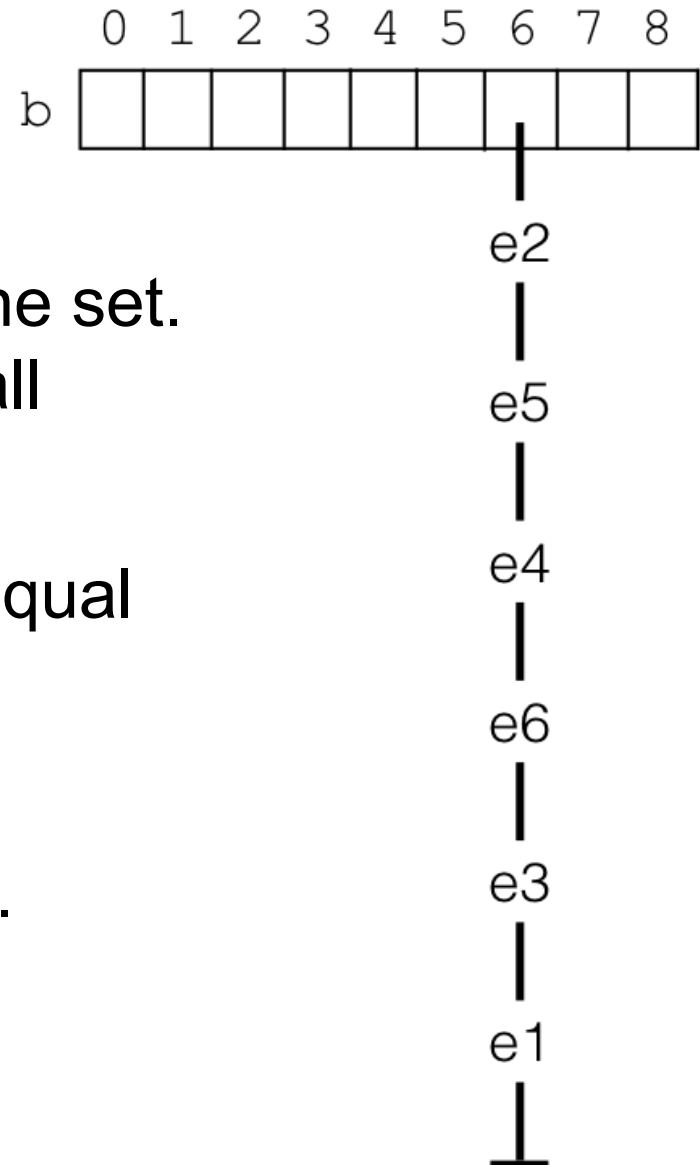
Suppose everything hashes to 0, so that $b[0..n-1]$ contains the set of elements and $b[n..]$ are all null.

In this case, operations **add**, **contains**, and **remove** all take time $O(n)$. That's the worst case.

Chaining: Expected time if load factor small: $O(1)$

34

EXAMPLE. 6 elements, table size 9, load factor $6/9$



Consider searching for e ---not in the set.
Find average length of chain over all possibilities.

e hashes to a number in $0..8$ with equal probability.

8 of the possibilities have length 0.
The other 1 possibility has length 6.

$$(8 \cdot 0 + 1 \cdot 6) / 9 = 6/9 \text{ (load factor)}$$

Chaining: Expected time if load factor small: $O(1)$

35

Example. 6000 elements,
table size 9000,
load factor $6/9$

0	8999
8999 nulls, 1 list of size 6000	

Find average length of chain over all possibilities.

e hashes to a number in 0..8999 with equal probability.

8999 of the possibilities have length 0.
The other 1 possibility has length 6000.

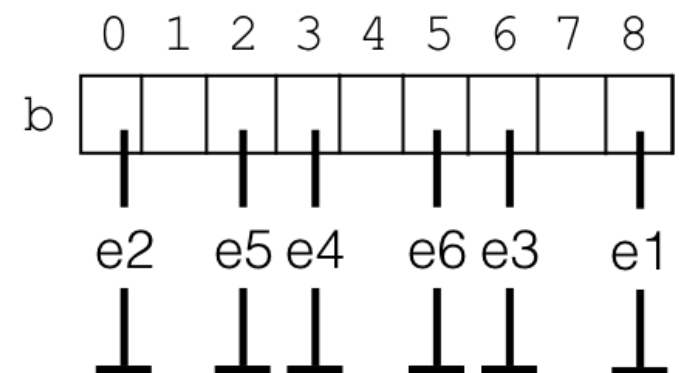
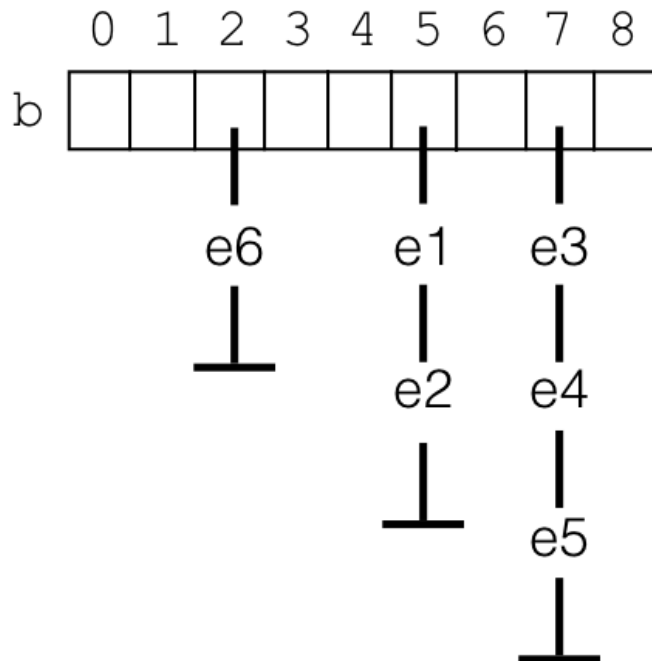
$$(8999 \cdot 0 + 1 \cdot 6000) / 9000 = 6/9 \text{ (load factor)}$$

Chaining: Expected time if load factor small: $O(1)$

36

Example: 6 elements, table size 9, load factor $6/9$

Consider *any* configuration of a set with load factor $6/9$.
The average chain length is the load factor: $6/9$



Average chain length: $6/9$

Chaining: Expected time if load factor small: $O(1)$

37

Searching for a value, whether in the set or not.

If the distribution of elements to buckets is sufficiently uniform, the average cost of a lookup depends only on the average number of elements per bucket.

That is: $(\text{size of set}) / (\text{size of array})$

That's the load factor!

Load factor .75: average of .75 elements per bucket

Load factor 1: average of 1 element per bucket

Load factor 2: average of 2 elements per bucket

Java HashMap uses chaining with load factor .75

Linear probing: Expected time, small load factor: $O(1)$

38

This analysis is more complicated, harder.
State without proof:

The number of probes (buckets examined) to insert a value in a hash table with load factor lf is

$$1 / (1 - lf)$$

Choose $lf = 1/2$ and get average number of probes: 2

Resizing



When the load factor gets too big, create a new array twice the size, move the values to the new array, and then use the new array going forward

YOU DID THIS IN A5, method `ensureSpace()`!

Collections class `ArrayList` does the same.

Collections classes `HashSet` and `HashMap` resize when the load factor becomes greater than `.75`, but you can change it.

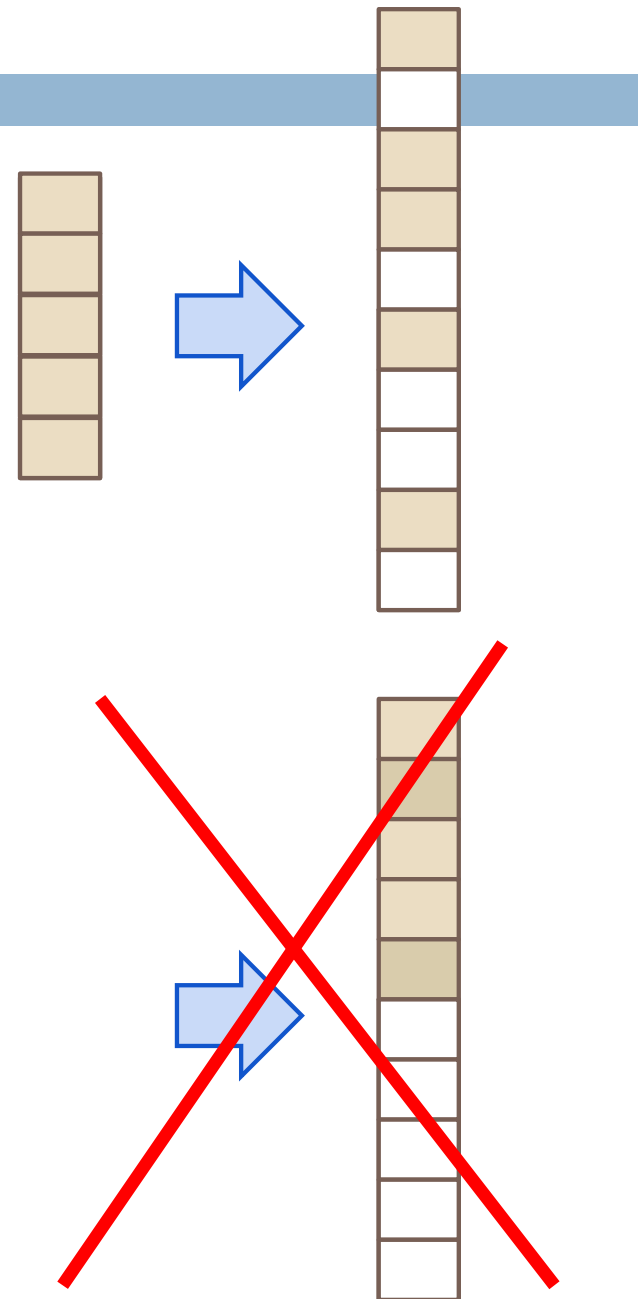
Resizing

Solution: ***Dynamic resizing***

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

index for an item is:
hash code mod table-size

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



Resizing takes constant amortized time

We bought a machine that makes fizzy water.
The machine cost \$100.

Say it cost \$100.00

AUTOMATIC CARBONATION

"touch button" activation



Make one glass of fizzy water:
glass cost \$100.00.

Make 100 glasses of fizzy water:
Each glass cost \$1.00.

Make 1,000 glasses:
Each glass cost 10 cents.

Amortizing cost of machine over
use of machine, over number of
operations “make a glass ...”.

image taken from sodastreamusa.com

Amortizing the cost of resizing

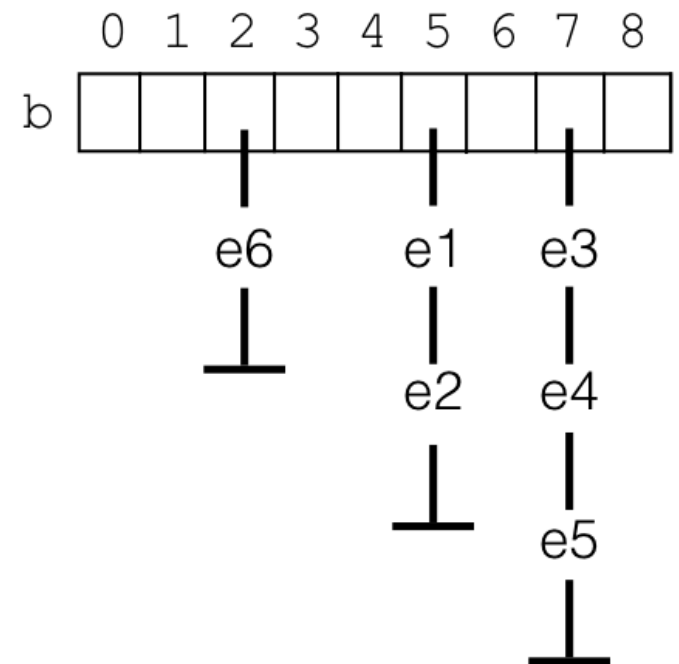
42

Each element of the array took at most constant time C (say) to add it to the set.

Double the size of the array:

Each element has to be rehashed into the new array, taking time at most C .

So we say that the time for each element is $2C$ —we amortize the cost of resizing over the time for the add operation.



Collision Resolution Summary

43

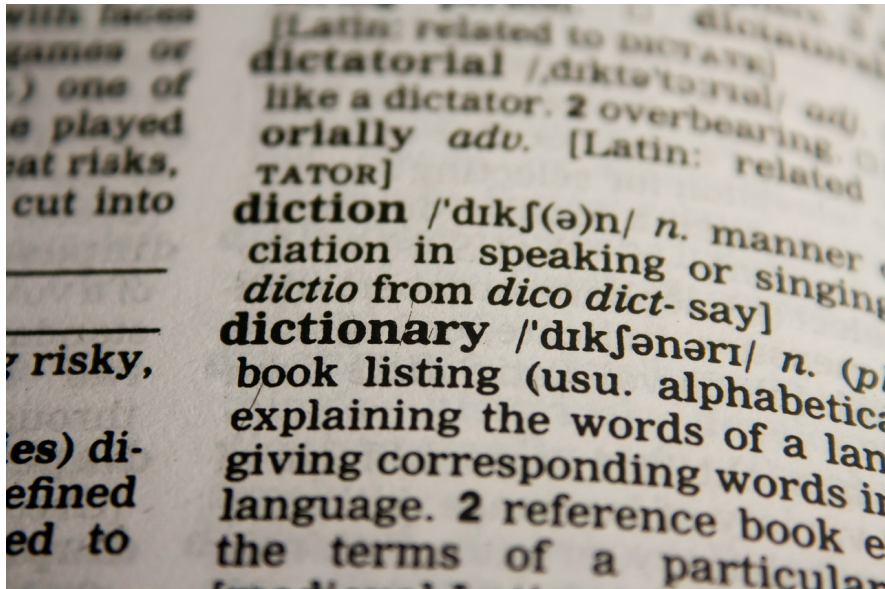
Chaining

- store entries in separate chains (linked lists)
- Uses more memory

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



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

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

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

HashMap in Java

45

- ❑ 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

46

- Network switches
- Distributed storage
- Database indexing
- Heaps with the ability to change a priority
- Index lookup (e.g. Dijkstra's shortest-path algorithm)
- Useful in lots of applications...