Recitation 7

Hashing

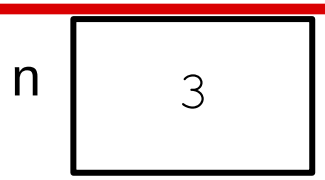
Set

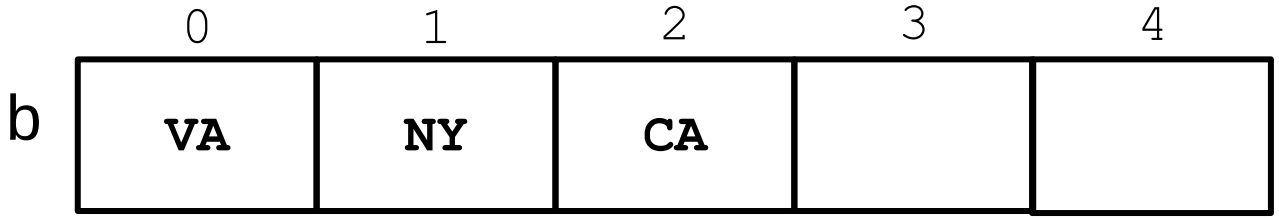
Set: collection of *distinct* objects

```
        Set<E>
add(E ob);
remove(E ob);
contains(E ob);
isEmpty()
size()
…    (a few more)
```

Implementing a set in an array

b[0..n-1] contains the values in the set





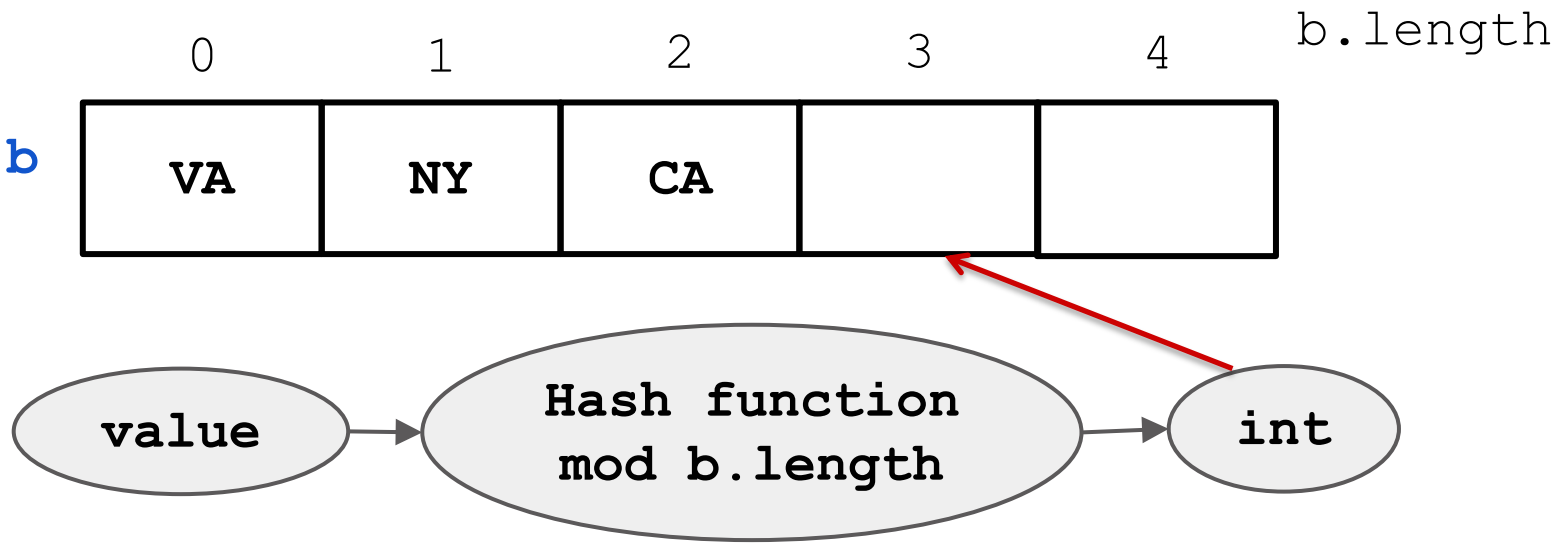
Have to search through the list *linearly* to find values

Have to shift all values down

method	expected time
`add`	*O(n)*
`contains`	*O(n)*
`remove`	*O(n)*

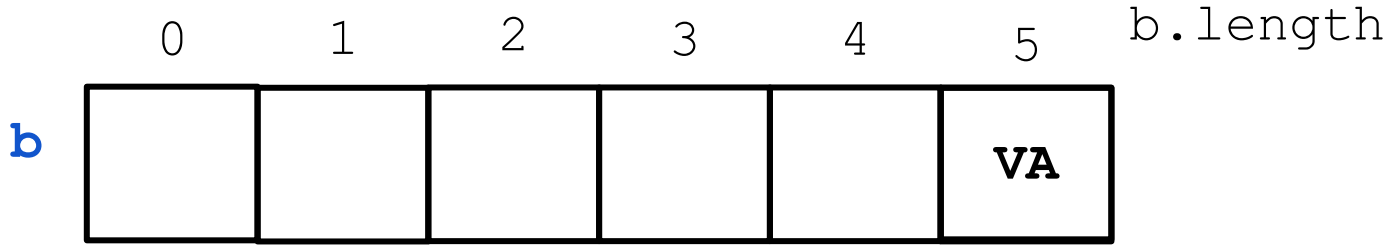
Hashing — an implementation of a Set

Idea: Use a hash function to tell where to put a value



Possible hash function for an object: its address in memory
(not always good, explain later)

Hashing



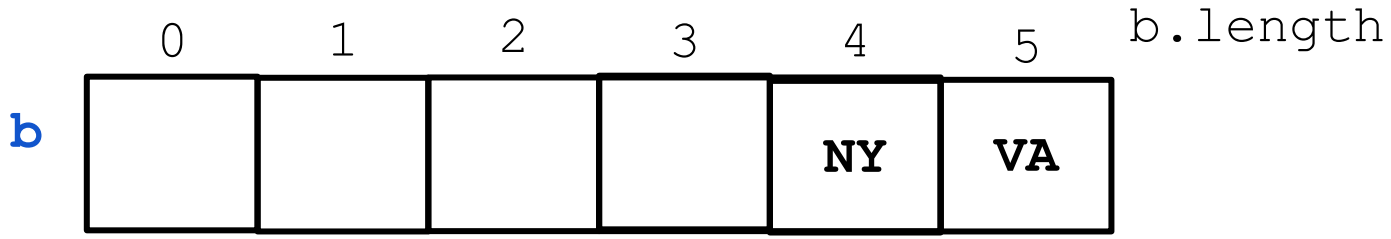
add("VA") can be done using

k= Math.abs(hashCode("VA")) % b.length;
if (b[k] == null) b[k]= "VA";

Suppose k is 5. This puts "VA" in b[5]

If b[k] != null?
Handle that later

Hashing



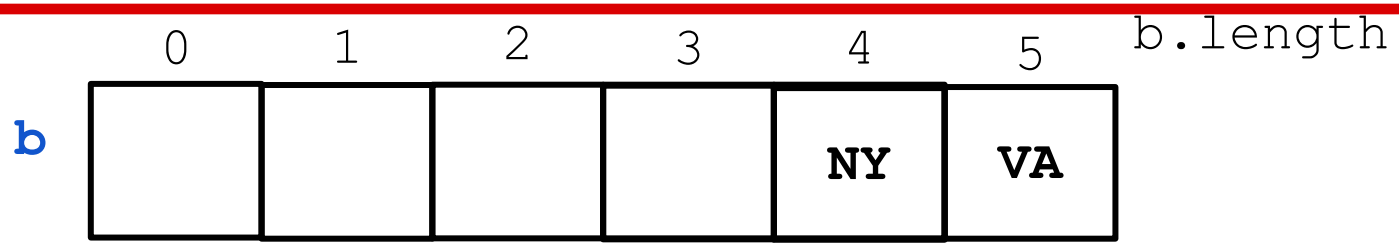
add("NY")

k= Math.abs(hashCode("NY")) % b.length;
if (b[k] == null) b[k]= "NY";

Suppose k is 4. This puts "NY" in b[4]

Collision Resolution

Collision resolution



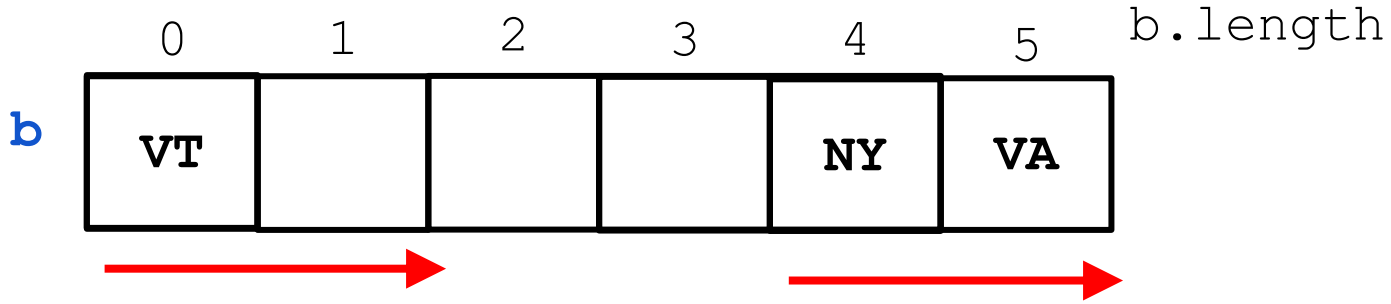
`add("VT")`

```
k= Math.abs(hashCode("VT")) % b.length;
if (b[k] == null)   b[k]= "VT";
```

Suppose k is 4. Can't place "VT" in b[4] because "NY" is already there

Two ways to solve collisions: Open addressing and chaining.
Do open addressing first

Open addressing: linear probing

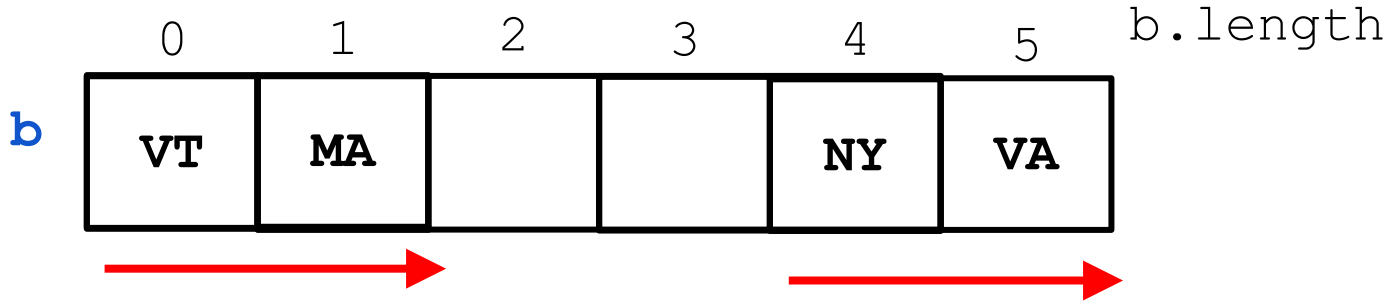


add("VT"). Suppose "VT" hashes to 4

Search in successive locations (with wraparound) for the first null element, and place "VT" there.

Here, look in b[4], b[5], b[0], and place "VT" in b[0].

Open addressing: linear probing

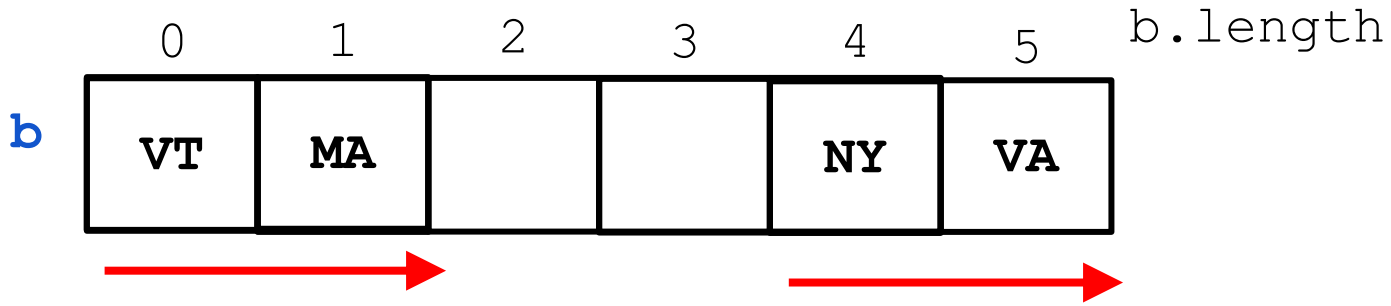


add("MA"). Suppose "MA" hashes to 4

Here, look in b[4], b[5], b[0], b[1] and place "MA" in b[1].

This took 4 probes to find a null element.
"probe": a test of one array element

Open addressing: linear probing



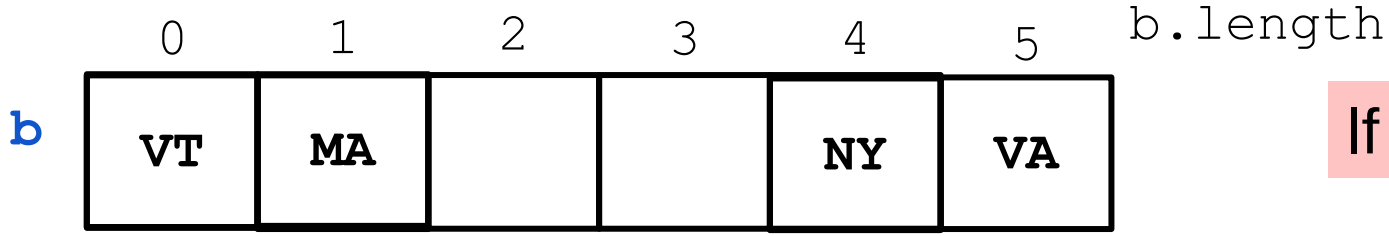
basic code for add(String s):

```
int k=  what  s  hashed to;
while (b[k] != null  &&  !b[k].equals(s))
        { k= (k+1) % b.length(); }

if (b[k] = = null) { b[k]=  s; } // if not null, s already in set
```

Making linear probing take expected constant time

Load factor lf: (# non-null elements) / b.length



lf = 4 / 6

Somebody proved:

Under certain assumptions about the hash function, the average number of probes used to add an element is **1 / (1 – lf)**

Making linear probing take expected constant time

Somebody proved:

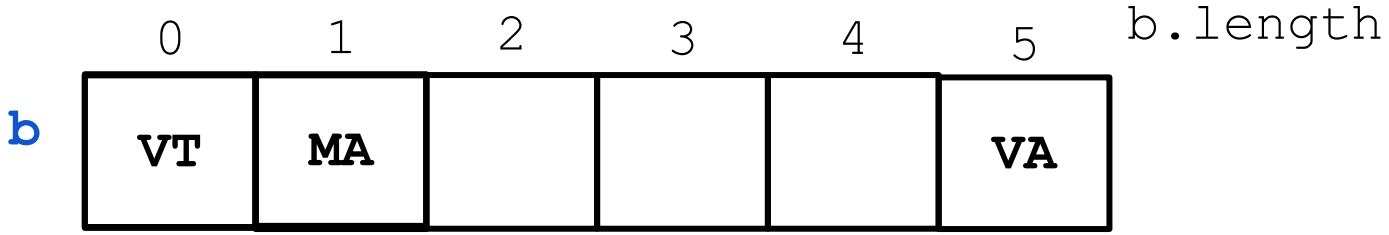
Under certain assumptions about the hash function, the average number of probes to add an element is **1 / (1 – lf)**

So if $lf \leq ½$, meaning at least half the elements are null, then the average number of probes is $\leq 1/(1/2) = 2$.

WOW! Make sure at least half the elements are null and expect no more than two probes!!! How can that be?

Making linear probing take expected constant time

Load factor lf: (# non-null elements) / b.length



If at least half the elements are null, expect no more than two probes !!!

Proof outside scope of 2110

Here's insight into it. Suppose half the elements are null. Then, half the time, you can expect to need only 1 probe.

Rehash: If the load factor becomes ≥ ½

If the load factor becomes ≥ ½, do the following:

1. Create a new empty array b1 of size 4*b.length
2. For each set element that is in b, hash it into array b1.
3. b= b1; // so from now on the new array is used

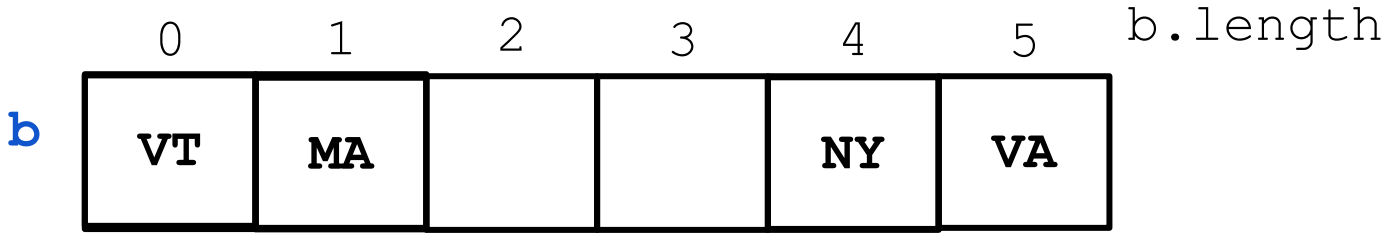
Suppose size of array goes from n to 4n. Then, can add more than n values before this has to be done again.
We can show that this does not increase the expected run time. We "amortize" this operation over the add operations that created the set.

What does "amortize" mean?

We bought a machine that makes fizzy water –adds fizz to plain water. Now, we don't have to buy fizzy water by the bottle. The machine cost $100.

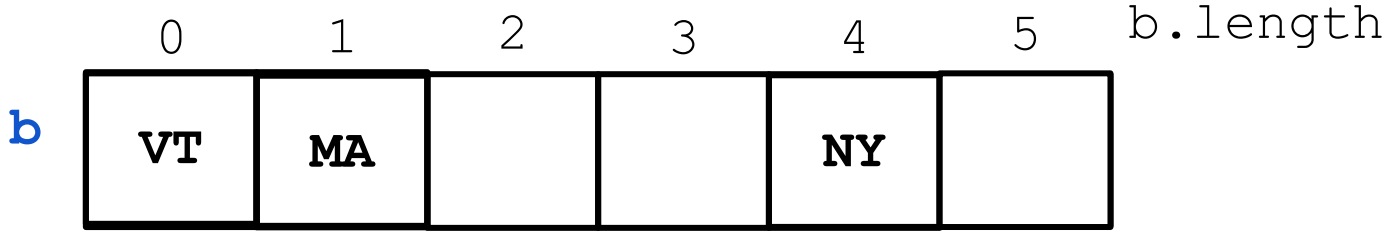
Use the machine to make one glass of fizzy water, that glass cost us $100.00.
Make 100 glasses of fizzy water? Each glass cost us $1.00.
Make 1,000 glasses? Each glass cost us10 cents.
I are amortizing the cost of the machine over the use of the machine, over the number of operations "make a glass …".

Deleting an element from the set



Does set contain “MA”?
“MA” hashes to 4. After probes of b[4], b[5], b[0], b[1], we say, yes, “MA’ is in the set.

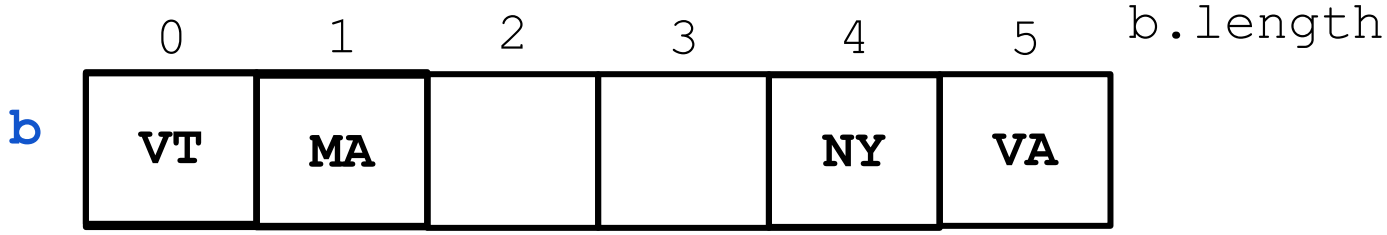
Deleting an element from the set



Does set contain “MA”?
“MA” hashes to 4. After probes of b[4], b[5], b[0], b[1], we say, yes, “MA’ is in the set.

Now suppose we delete “VA” from the set, by setting b[5] to null.
Now ask whether the set contains “MA”. Two probes say no, because the second probe finds null!!!

Deleting an element from the set

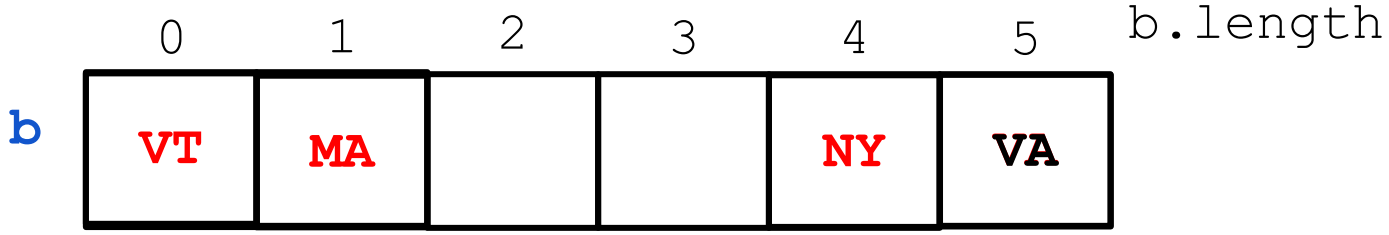


Therefore, we can't delete a value from the set by setting its array element to null. That messes up linear probing.

Instead, in Java, use an inner class for the array elements, with two fields:

1. String value; // the value, like "VT"
2. boolean isInSet; // true iff value is in the set

Deleting an element from the set



Instead, in Java, use an inner class for the array elements, with two fields:

1. String value; // the value, like "VT"
2. boolean isInSet; // true iff value is in the set

Above: red string means its isInSet field is true.
To delete "VA", set its isInSet field to false

Inner class HashEntry

```
class HashSet<E> {
      LinkedList<HashEntry<E>>[] b;

      private class HashEntry<E> {
          private E value;
          private boolean isInSet;
      }
}
```

inner class to contain value and whether it is in the set
Class is private ---the user knows nothing about it

Summary for open addressing –linear probing

1. Each non-null b[i] contains an object with two fields: a value and boolean variable isInSet.
2. add(e). Hash e to an index and linear probe. If null was found, add e at that spot. If e was found, set its isInSet field to true.
   If load factor >= ½, move set elements to an array double the size.
3. Remove(e). Hash e to an index and linear probe. If null was found, do nothing. If e was found, set its isInSet field to false.
4. Contains(e). Hash e to an index and linear probe. If e was found and its isInSet field is true, return true; otherwise, return false.

DEMO. We have a complete implementation of this.

Hash Functions

Class Object contains a function hashCode().
The value of C.hashCode() is the memory address where the object resides.

You can override this function in any class you write. Later slides discuss why one would do this.

For primitive types, you have to write your own hashCode function.

On the next slides, we discuss hash functions.

Requirements

Hash functions MUST:

- have the same hash for equal objects
  - In Java: if `a.equals(b)`, then
    `a.hashCode() == b.hashCode()`
  - if you override equals and plan on using object in a `HashMap` or `HashSet`, override `hashCode` too!

- be deterministic
  - calling `hashCode` on the same object should return the same integer
    - important to have immutable values if you override equals!

Good hash functions

- As often as possible, if !a.equals(b), then a.hashCode() != b.hashCode()
  - this helps avoid collisions and clustering
- Good distribution of hash values across all possible keys
- FAST. add, contains, and remove take time proportional to speed of hash function

A bad hash function won't break a hash set but it could seriously slow it down

String.hashCode()

Don’t hash long strings, not O(1) but O(length of string)!

```
/** Return a hash code for this string.
 *  Computes it as
 *    s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 *  using int arithmetic.
 */
public int hashCode() { ... }
```

Designing good hash functions

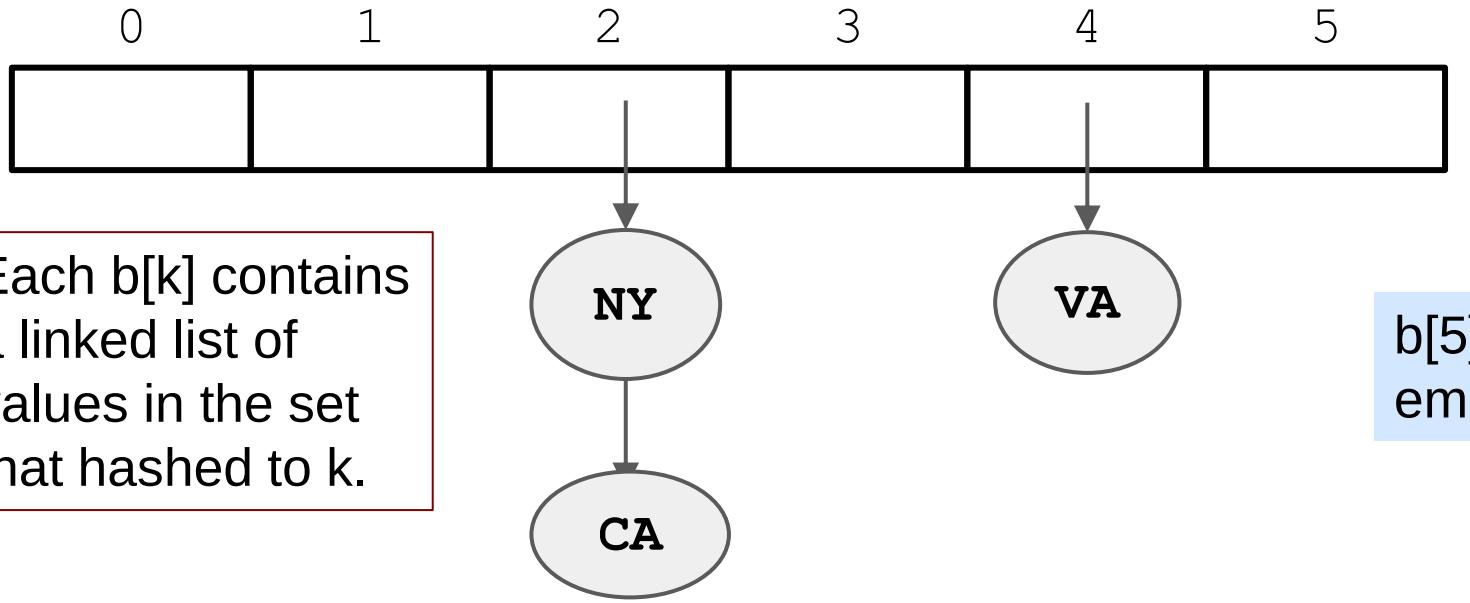
```
class Thingy {
      private String s1, s2;

      public boolean equals(Object obj) {
            return s1.equals(obj.s1) &&
                  s2.equals(obj.s2);
      }
   public int hashCode() {
      return 37 * s1.hashCode() + 97 * s2.hashCode();
   }
}
```

Collisions: Chaining

an alternative to open addressing (probing)

Chaining definition



Each b[k] contains a linked list of values in the set that hashed to k.

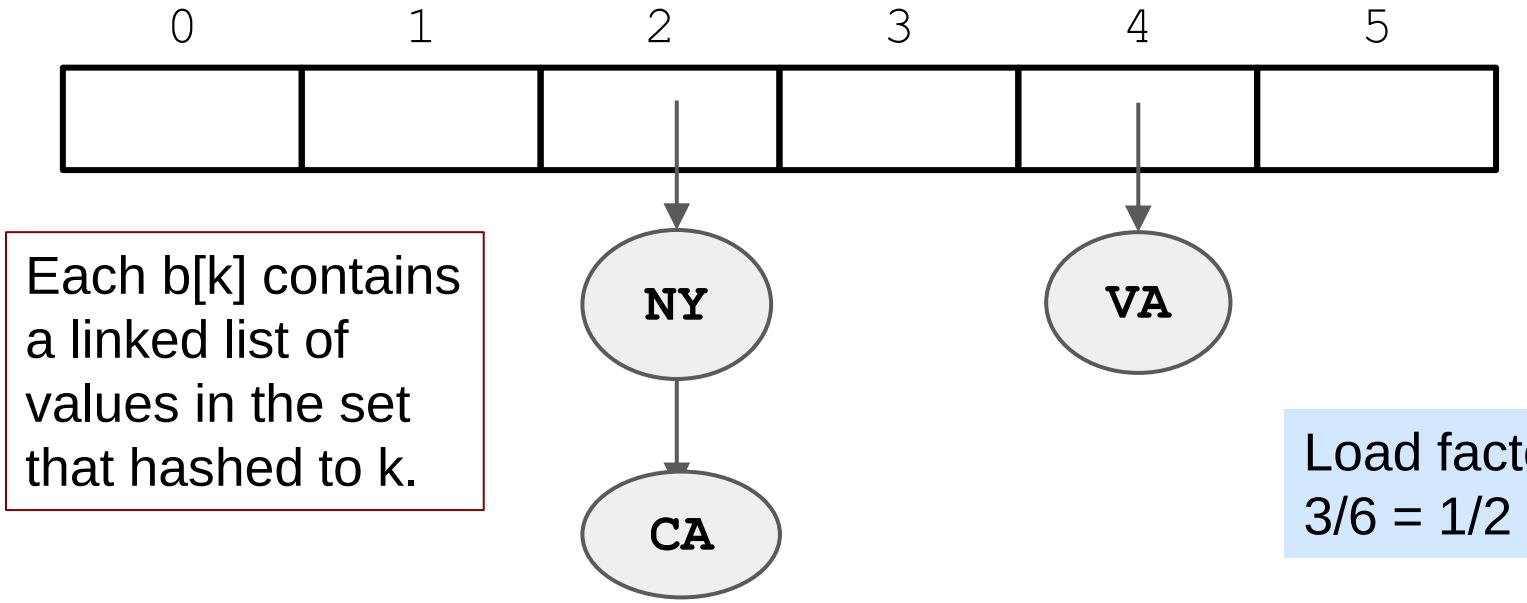
b[5] is an empty list

add(e): hash e to some k. If e is not on linked list b[k], add it to the list

remove(e): hash e to some k. If e is on linked list b[k], remove it

You can figure out other operations yourself.

Chaining



Each b[k] contains a linked list of values in the set that hashed to k.

Load factor is 3/6 = 1/2

The load factor: (number of values in list) / size of array
It must be kept under ½, as with open addressing

Linear probing versus quadratic probing

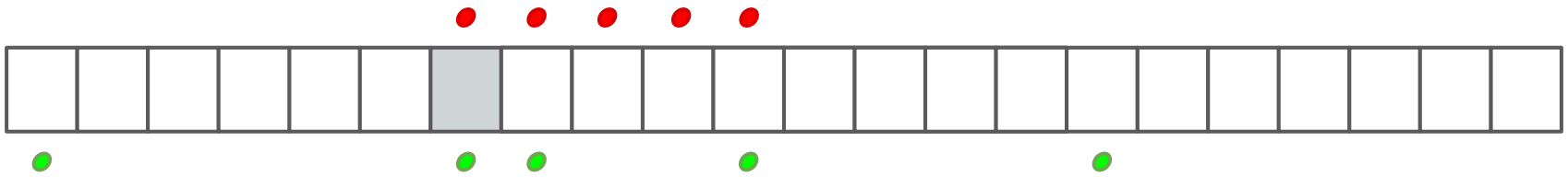
Linear vs quadratic probing

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, $i+1^2$, $i+2^2$, $i+3^2$ . . .

For quadratic probing, the size of the array should be a prime. Someone proved that then, every single array element will be covered.

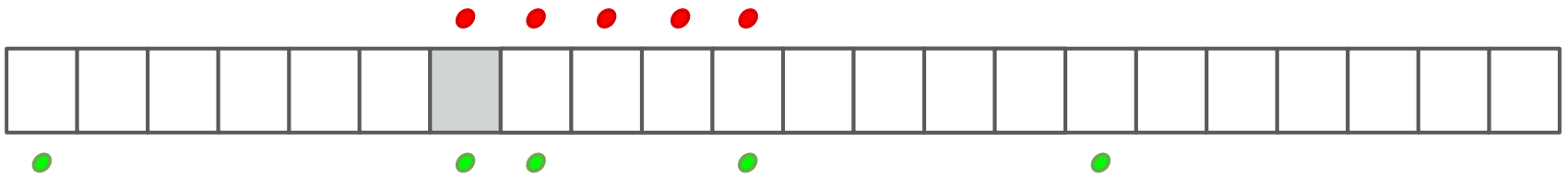


Why use quadratic probing

linear probing:
i, i+1, i+2, i+3 . . .

quadratic probing:
i, $i+1^2$, $i+2^2$, $i+3^2$ . . .

Collisions can lead to clustering: many full elements in a row. Quadratic probing spreads the values out more, leading to less clustering than with linear probing.



Big O!

Runtime analysis

	Open Addressing	Chaining
Expected	O(hash function) (since load factor kept < ½)	O(hash function) (since load factor kept < ½)
Worst	O(n) (no null between values)	O(n) (all values in one linked list)

Amortized runtime

Insert n items: n + 2n (from copying) = 3n inserts → O(3n) → O(n)
Amortized to constant time per insert

	Copying Work
Everything has just been copied	n inserts
Half were copied in previous doubling	n/2 inserts
Half of those were copied in doubling before previous one	n/4 inserts
...	...
Total work	n + n/2 + n/4 + … ≤ 2n

Limitations of hash sets

1. Due to rehashing, adding elements may take O(n)
    a. not always ideal for time-critical applications

1. No ordering among elements, very slow to find nearby elements

Alternatives (out of scope of the course):

1. hash set with incremental resizing prevents O(n) rehashing

1. self-balancing binary search trees are worst case O(log n) and keep the elements ordered

Hashing Extras

Hashing has wide applications in areas such as security

- cryptographic hash functions are ones that are very hard to invert (figure out original data from hash code), changing the data almost always changes the hash, and two objects almost always have different hashes

- md5 hash: `md5 filename` in Terminal

By	Size	MD5
erc73	1.449 MB	13188ff26829b7cc4f4a45b84ee6deb7