Recitation 7

Hashing

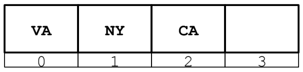
Sets

```
Set<E>
add(E e);
remove(Object o);
contains(Object o);
size()
```

Set: collection of *distinct* objects

How to implement a set?

Array List of *values*?



Method	Runtime
`add`	*O(n)*
`contains`	*O(n)*
`remove`	*O(n)*

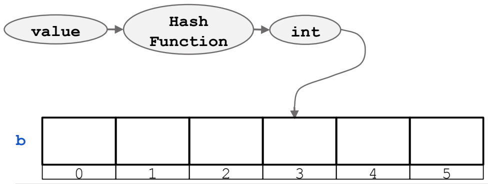
Have to search through the list *linearly* to find the values → `add`, `contains`

Have to shift all values down → `remove`

Hashing 101

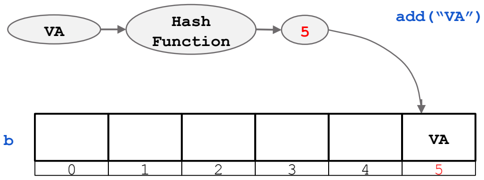
Hashing — an implementation of a Set

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



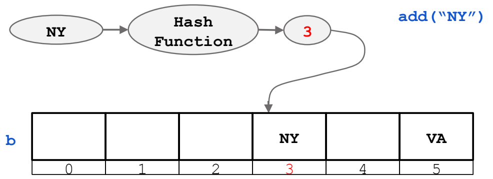
Hashing

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



Hashing

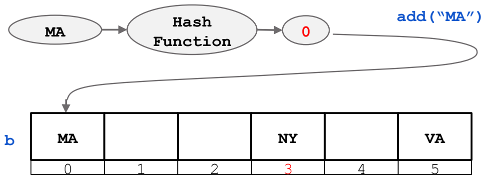
Idea: finding an element in an array takes constant time when you know which index it is stored in



Load factor: `b`'s saturation

Load factor:

$$\lambda = \frac{\textit{\# of entries}}{\textit{length of array}} = \frac{3}{6}$$



We can hash any type of object!

Every object in Java has this method.

Default behavior is its object's memory address.

```
class Point {
      int x;
      int y;

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

Remainder Operator!

What if hashCode returns an int out of the array's bounds?

```
int hashInBounds(Object val) {
      return Math.abs(val.hashCode() % b.length);
}
```

For all operations, start by hashing to a valid index

Basic set operations with hashing

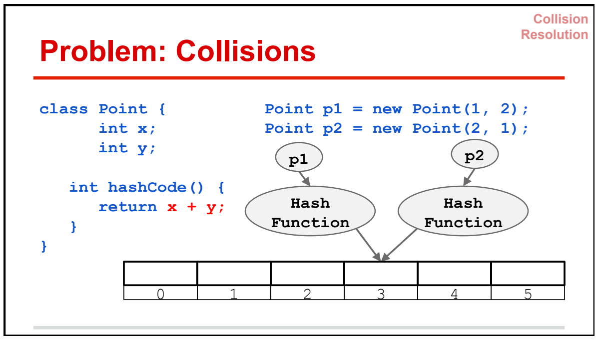
```
add(val) {
      b[hashInbounds(val)]= val;
}
remove(val) {
      b[hashInbounds(val)]= null;
}
contains(val) {
   return b[hashInbounds(val)]
            != null;
}
```

Note: these are ***very*** simplified versions!

Operations take time proportional to hash function. Constant with respect to size of the array!

Collisions are a big problem: 2 vals hash to same index!

Collision Resolution



Collision Resolution

Problem: Collisions

```
class Point {
      int x;
      int y;

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

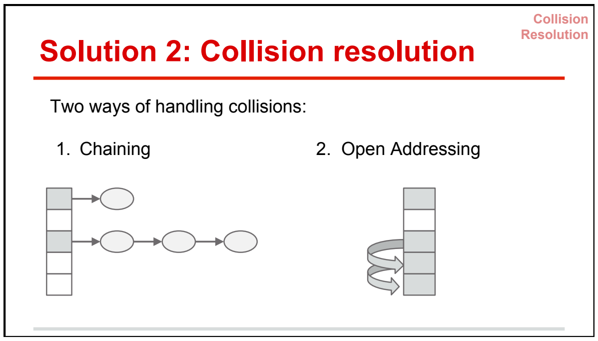
```
Point p1 = new Point(1, 2);
Point p2 = new Point(2, 1);
```

Collision Resolution

Solution 1: Perfect hash function

Map each value to a different index in the hash table

Impossible in practice
- don't know the size of the array
- Number of possible values far far exceeds the array size
- no point in a perfect hash function if it takes O(n) to compute



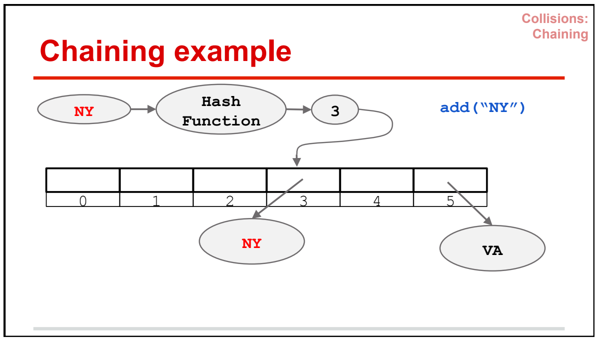
Collision Resolution

Solution 2: Collision resolution

Two ways of handling collisions:

1. Chaining
2. Open Addressing

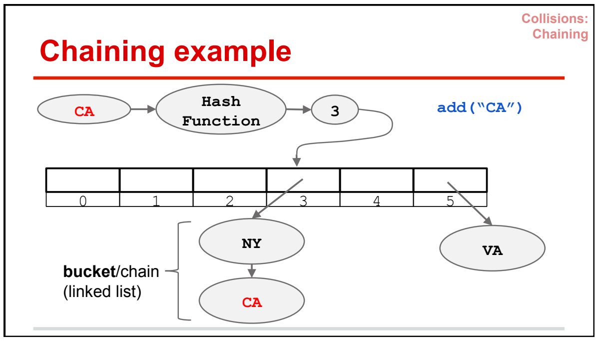
Collisions: Chaining



Collisions: Chaining

Chaining example

`add("NY")`



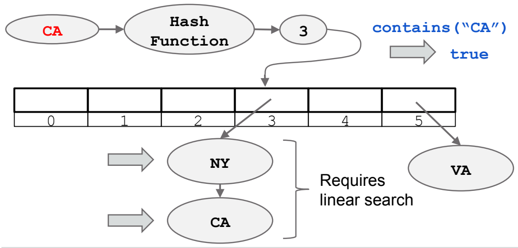
Collisions: Chaining

Chaining example

`add("CA")`

bucket/chain (linked list)

Chaining example



Inner class HashEntry

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

        private class HashEntry<V> {
                V value;
        }
}
```

inner class to store value

Set operations

For `add, contains, remove` always start by finding correct bucket:

- `b[hashInBounds(value)]`

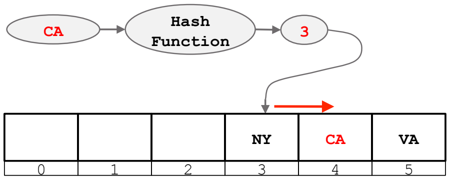
```
add(value)
        1. If bucket already contains value, do nothing
        2. Else add new HashEntry to bucket
contains(value)
        1. If bucket contains value, return true
        2. Else return false
remove(value)
        1. If bucket contains value, remove entry from
list
```

Collisions:
Open Addressing

Open addressing example

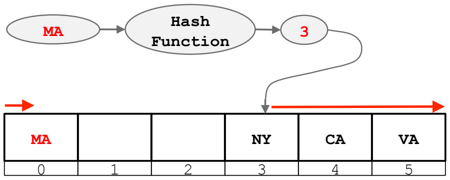
`add("CA")`

probing: Find another available space



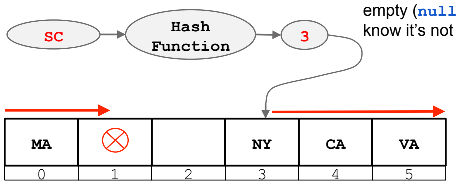
Open addressing example

`add("MA")`



Open addressing example

`contains("SC")`



How far do we search? Once we reach an empty (`null`) cell, we know it's not there.

Finding where a key belongs

Keep searching until we hit `null` or we find the `value` in question

```
int getPosition(val) {
   int i = hashInBounds(val);
   while (b[i] != null && !val.equals(b[i].val)) {
      i = (i+1) % b.length;
   }
   return i;
}
```

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

Efficiency of linear probing

Average number of probes

$$= \frac{1}{1-\lambda} = \frac{1}{1-\frac{\#\text{ of entries}}{\text{length of array}}} = \frac{1}{\frac{\text{null entries}}{\text{length of array}}} = \frac{\text{length of array}}{\text{null entries}}$$

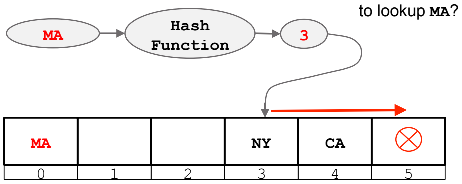
Array half full? `add(value)` expected to need only 2 probes! Wow!

Beats linear search!

Deleting elements

`contains("MA")` ⇨ `false`

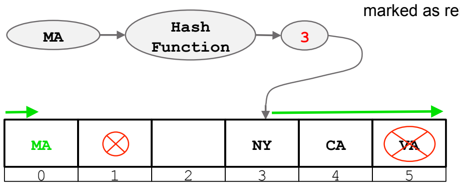
What happens if we remove **`VA`** and then try to lookup **`MA`**?



Deleting elements

`contains("MA")` ⇨ `true`

Solution: The **`VA`** entry is still there, but marked as removed



Deleting elements

```
class HashSet<V> {
       HashEntry<V>[] b;

       private class HashEntry<V> {
              V value;
              boolean isInSet= true;
       }
}
```

Set `isInSet` to `false` to remove it

Set operations

For `add, contains, remove,` always start by finding correct index using probing: `pos = getPosition(key)`

```
add(value)
      1. If b[pos] is null, add new HashEntry at pos
      2. Else mark isInSet as true
contains(value)
      1. Return b[pos] != null && b[pos].isInSet
remove(value)
      1. If b[pos] is not null and isInSet is true,
            mark isInSet as false
```

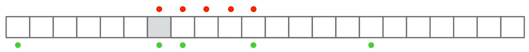
Linear vs quadratic probing

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

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



Collision resolution summary

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

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

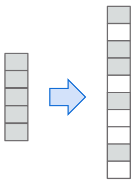
Rehashing

Resizing

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

O(1) → O(n) operations

Solution: ***Dynamic resizing***
- reinsert / **rehash** all elements to an array *double* the size.
  - Now is the time where we remove the entries where `!b[pos].isInSet`
- **Why not** simply copy into first half?



Load factor

Load factor →

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

Rehashing happens when λ reaches ***load factor threshold***

Big O!

Big O of Hashing

Runtime analysis

	Chaining	Open Addressing
Expected	O(hash function) + O(load factor)	O(hash function) + $O\left(\frac{length\ of\ array}{\#\ of\ null\ slots}\right)$
Worst	**O(n)** (all elements in one bucket)	**O(n)** (array almost full)

Big O of Hashing

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

Hash Functions

Hash Functions

Requirements

Hash functions MUST:

- have the same hash for two 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!

Hash Functions

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 are 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 very long strings, not O(1) but O(length of string)!

```
/** Returns 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();
   }
}
```

Limitations of hash sets

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

2. 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

2. 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