

PG4200: Algorithms And Data Structures

Lesson 06: Hash Maps and Sets

Dr. Andrea Arcuri

Hash Function

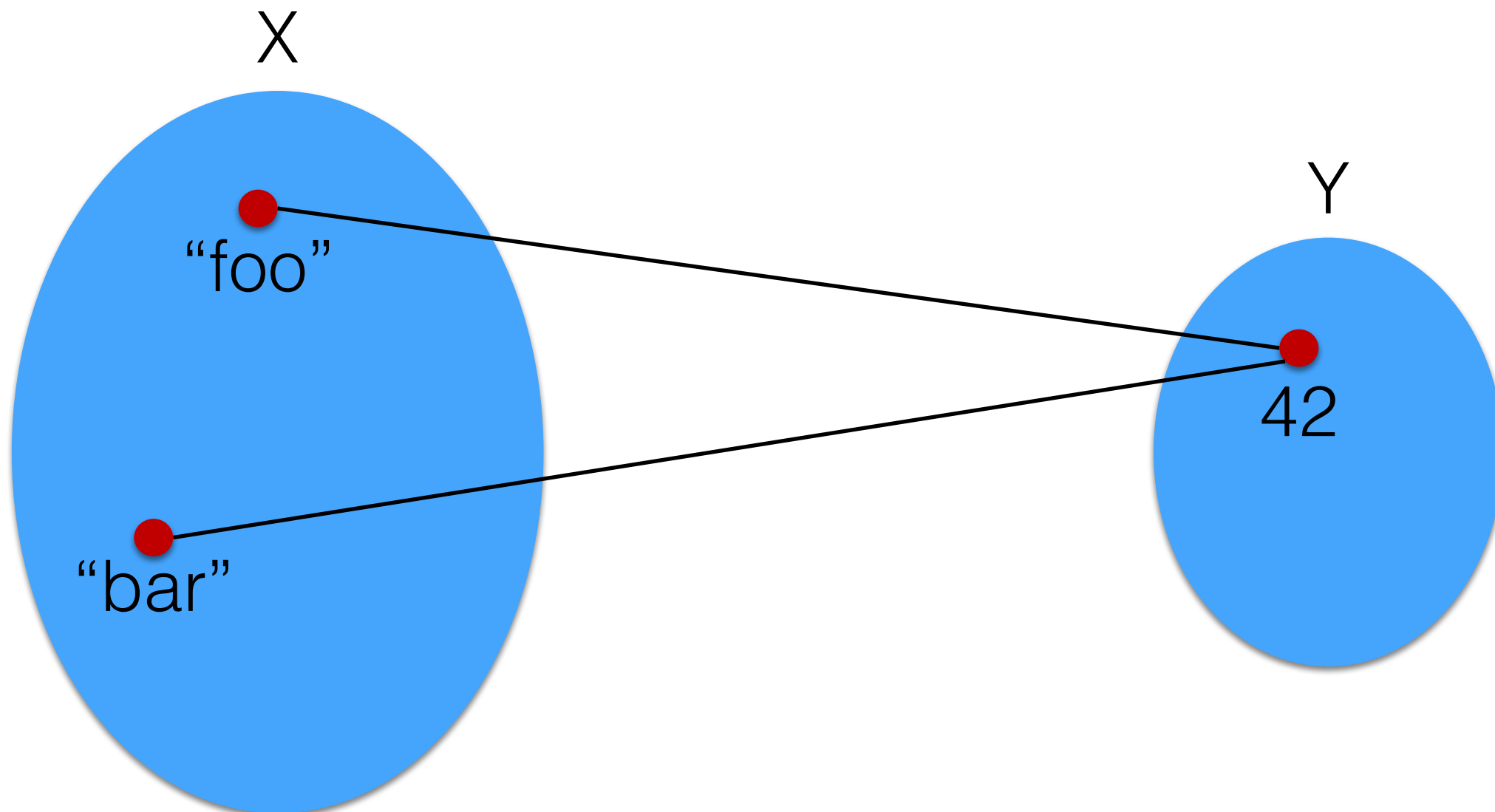
- A function that maps data from an arbitrary size to a specific size
 - eg, mapping strings to a int
- $h(x)=y$, mapping from domain X to a value in domain Y
- $|X|$ is often much larger than $|Y|$

Hash Properties

- *Deterministic*: for a given input x' , should always get the same output y'
- *Uniform*: mapping from X to Y should be ideally spread uniformly over Y ,
 - ie the number of elements in X that map to a specific y' should be close to $|X|/|Y|$
- *Performance*: either fast (in this course) or slow (security, eg hashing of passwords)

Collisions

- If $|X| > |Y|$, you cannot avoid $h(x')=h(x'')$, two different values in X mapping to the same value in Y
- Ideally, if uniform, no more than $|X|/|Y|$ collisions per element



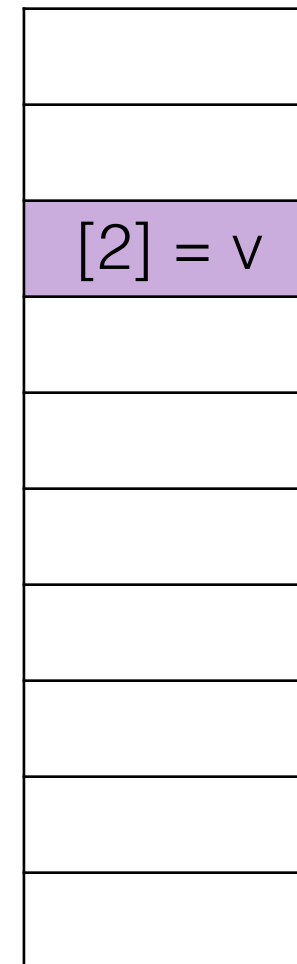
Hash Maps

- Still a map from a K key to a V value
- No requirement on ordering of K keys, just being able to compute an *hash* of it
- In Java, all objects inherits from *java.lang.Object*, which defines a *hashCode()* method
- Hash code used as an index for an internal array

Example

- $put("foo", v)$
- $h("foo")=42$
- $h("foo") \% 10 = 2$
- Benefit: operations (insert/search/etc) have cost due to hash independent of size N of the collection

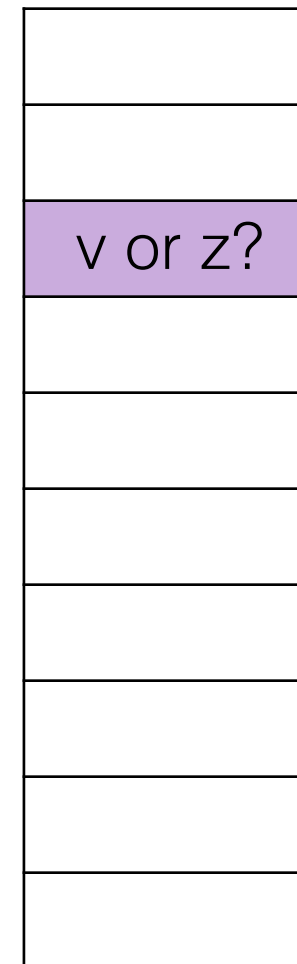
Internal array buffer of size $M=10$



What About Collisions?

- *put*("foo", v)
- *put*("bar", z)
- $h(\text{"foo"}) = h(\text{"bar"})$
 - ie, collision due to same hash
- $h(\text{"foo"}) \% 10 = 2$
- What to do?

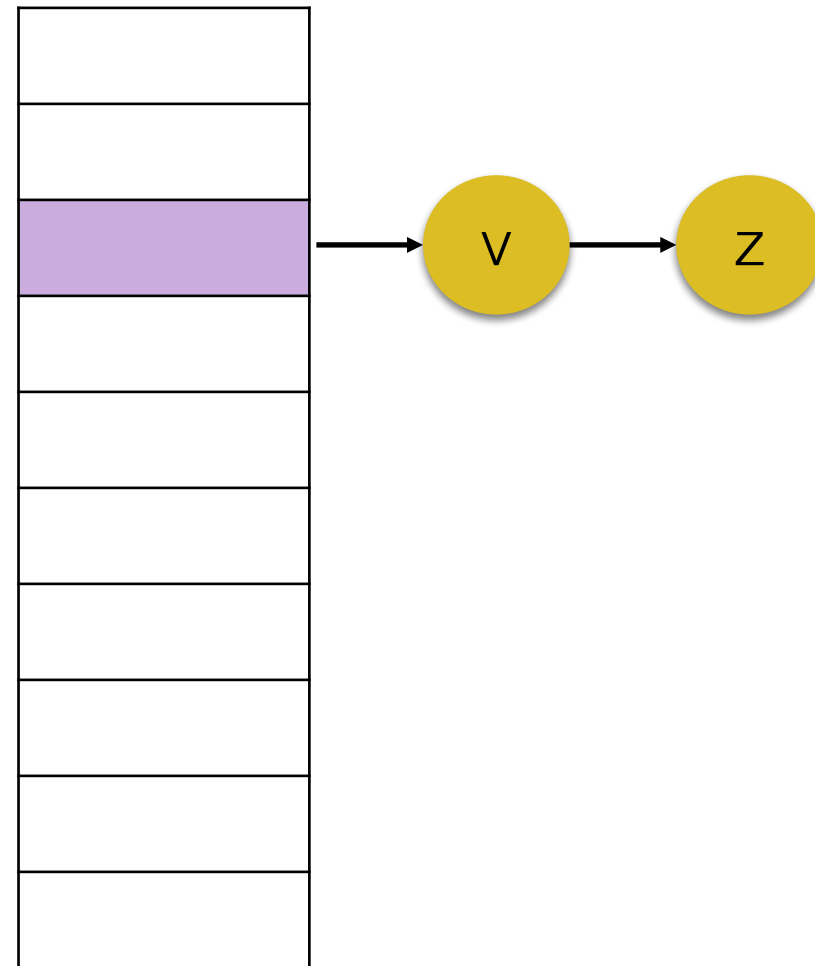
Internal array buffer of size $M=10$



Different Strategies

- *put*("foo", v)
- *put*("bar", z)
- $h(\text{"foo"}) = h(\text{"bar"})$
 - ie, collision due to same hash
- Use list at each position sharing same hash

Internal array buffer of size $M=10$



Cost

- Worst case: **$O(N)$** if all elements end up in same “bucket” (ie same value for $h() \% M$), the map would be equivalent to a list
 - operations to search on list would be $O(N)$, albeit insert would be $O(1)$
- But, **if** M large enough compared to N , and hash function is uniform enough, you can have a **$O(1)$** cost in many cases
 - even if you have some collisions, it will not be a problem, as you would have a small number of elements in the list

Hash or RBT?

- Hash Maps is the most popular and widely used
- If you know how much data you'll insert at most, can choose a good large enough M
- So in most cases, we are in $O(1)$ Hash vs $O(\log N)$ RBT
- But Hash can be $O(N)$ in worst case, vs RBT **guarantees** $O(\log N)$ in all cases
 - eg, in critical systems where you **MUST** guarantee a response within a certain amount of time, might want to use RBT
- Hash does not need ordering of keys

Set

- In mathematics, a *set* is a collection of elements where:
 - 1) *ordering is not important*: ie $\{1,2,3\}$ is equivalent to $\{2,3,1\}$
 - 2) *no repetitions*: ie $\{1,2\}$ is the same as $\{2,1,1,2,2,1,1,2,1\}$
- How to implement a Set in Java?
- Easy: use an internal $\text{Map}\langle K, V \rangle$ where your values in the set are the keys K , and you just ignore the values V

Keys and Immutability

- *Immutable Object*: an object whose state cannot be changed once created
- Example: Strings are immutable
 - eg, concatenation with + and methods like *toUpperCase()* and *substring()* do NOT change the String, but rather *create* a NEW one
- Keys in a Map/Set **MUST** be *immutable*... why?

Different Hash

- `Foo foo = new Foo();`
- `set.add(foo);`
- `assertTrue(set.contains(foo));`
- `// h(foo) = 42 , 42 % M = 2`
- `foo.setSomeVariable(...);`
- `// h(foo) = 55 , 55 % M = 5`
- `assertFalse(set.contains(foo));`



foo
miss

java.lang.Object

- *Object* does define two methods: *hashCode()* and *equals()*
- Those methods will depend on the internal fields of the object
- *Important*: if two objects are equals, then they **MUST** have same hash code
 - $A.equals(B)$ implies $A.hashCode() == B.hashCode()$
 - The vice-versa is not necessarily true, ie $A.hashCode() == B.hashCode()$ does not imply $A.equals(B)$, although that could happen
- What if constraint is not satisfied? Expect weird bugs when using maps and sets...

Homework

- Study Book Chapter 3.4 and 3.5
- Study code in the *org.pg4200.les06* package
- Do exercises in *exercises/ex06*
- Extra: do exercises in the book