L6: Hashing, HashMap, HashSet

Alex Steiger
CompSci 201: Spring 2024
1/31/24

1

Announcements, Coming up

- Today, Wednesday, 1/31
  - APT 2 due
- Monday, 2/5
  - Project 1: NBody due (future projects will be 2 week)
  - Project 2: Markov out (due in 2 weeks)
- Next Wednesday, 2/8
  - APT 3 due

2

Finishing `Maps` API

3

Map pairs keys with values

- Like an **address book**, lookup the value (address) of a key (person). Like a dictionary in Python.

Keys	Values
Bob	101 E. Main St.
Naomi	200 Broadway
Stavros	121 Durham Ave.

- Map is an interface, must have methods like:
  - `put(k, v)`: Associate value `v` with key `k`
  - `get(k):` Return the value associated with key k
  - `containsKey(k)`: Return true if key `k` is in the Map

4

Implementations of Map

```java
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
```

Two major implementations:

- `HashMap`: Very efficient `put`, `get`, `containsKey`
- `TreeMap`: Nearly as efficient, keeps ***keys*** sorted by their "*natural ordering*"

`Map<KEY_TYPE, VALUE_TYPE>`

Create a `TreeMap` to implement this `Map`

```java
Map<String, String> addressBook = new TreeMap<>();
addressBook.put("Bob", "101 E. Main St.");
addressBook.put("Naomi", "200 Broadway");
addressBook.put("Xi", "121 Durham Ave.");
System.out.println(addressBook);
```

Sorted by keys due to `TreeMap`

```
{Bob=101 E. Main St., Naomi=200 Broadway, Xi=121 Durham Ave.}
```

5

Check before you get

If you call `.get(key)` on a key not in the map, returns `null`, can cause program to crash.

```java
Map<String, Integer> myMap = new HashMap<>();
int val = myMap.get("hi");
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.lang.
Integer.intValue()" because the return value of "java.util.Map.get(Object)" is null
```

Instead, check first with `.containsKey()`.

```java
Map<String, Integer> myMap = new HashMap<>();
if (myMap.containsKey("hi")) {
    int val = myMap.get("hi");
}
```

6

Adding "default" values

Often want a "default" value associated with new keys (examples: 0, empty list, etc.). Two options:

- `.putIfAbsent(key, val)`
- Check if does not contain key before put

```
6     Map<String, Integer> myMap = new HashMap<>();
7
8     myMap.putIfAbsent("hi", 0);
9
10    // Equivalent to line 8
11    if (!myMap.containsKey("hi")) {
12        myMap.put("hi", 0);
13    }
```

7

Updating maps

Immutable values:

- `.get()` returns a *copy of the value.*
- Must use `.put()` again to update.

Mutable values (e.g. collections)

- `.get()` returns *reference to collection.*
- Update the collection directly.

```
Map<String, Integer> myMap = new HashMap<>();
myMap.put("hi", 0);
int currentVal = myMap.get("hi");
myMap.put("hi", currentVal + 1);
```

```
Map<String, List<Integer>> otherMap = new HashMap<>();
otherMap.put("hi", new ArrayList<>());
otherMap.get("hi").add(0);
```

8

Counting with a `Map`

In this example we count how many of each character occur in `message`.

```
String message = "computer science is so much fun";
char[] messageCharArray = message.toCharArray();
TreeMap<Character, Integer> charCounts = new TreeMap<>();
for (char c : messageCharArray) {
    if (!charCounts.containsKey(c)) {
        charCounts.put(c, 1);
    }
    else {
        int currentVal = charCounts.get(c);
        charCounts.put(c, currentVal + 1);
    }
}
System.out.println(charCounts);
```

```
{ =5, c=4, e=3, f=1, h=1, i=2, m=2, n=2, o=2, p=1, r=1, s=3, t=1, u=3}
```

Check if we have *not* seen c yet

Else get current value and increase

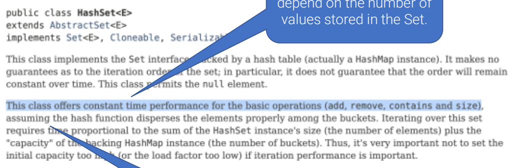
Comes in order because using `TreeMap`

9

HashSet/HashMap Implementation

14

HashSet/Map efficiency



```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializa
```

This class implements the Set interface cked by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration orde the set; in particular, it does not guarantee that the order will remain constant over time. This class rmits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires t proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of th backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too (or the load factor too low) if iteration performance is important.

Java API documentation

Constant time = does not depend on the number of values stored in the Set.

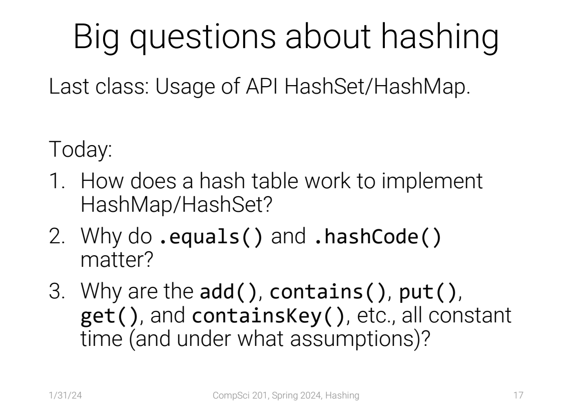
Under assumptions…

15

Aside: Does constant time lookup (`contains()`, `get()`, etc.) matter?

- Social media: When you login, server needs to lookup to display the correct page for you.
  - Billions of accounts! Look it up in a List? NO! Constant time lookup with hashing.
- Routing/directions application: Need to lookup roads from a given intersection.
  - How many possible roads? Search through a list? NO! Constant time lookup with hashing.
- Could go on!

16



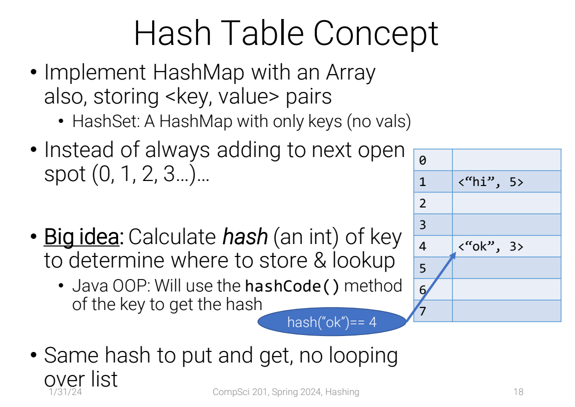
Big questions about hashing

Last class: Usage of API HashSet/HashMap.

Today:

1. How does a hash table work to implement HashMap/HashSet?
2. Why do `.equals()` and `.hashCode()` matter?
3. Why are the `add()`, `contains()`, `put()`, `get()`, and `containsKey()`, etc., all constant time (and under what assumptions)?

17



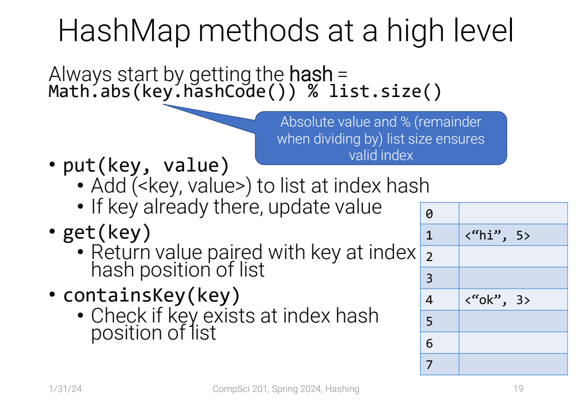
Hash Table Concept

- Implement HashMap with an Array also, storing <key, value> pairs
  - HashSet: A HashMap with only keys (no vals)
- Instead of always adding to next open spot (0, 1, 2, 3…)…
- <u>Big idea</u>: Calculate *hash* (an int) of key to determine where to store & lookup
  - Java OOP: Will use the `hashCode()` method of the key to get the hash
- Same hash to put and get, no looping over list

0	
1	<"hi", 5>
2	
3	
4	<"ok", 3>
5	
6	
7	

hash("ok")== 4

18



HashMap methods at a high level

Always start by getting the **hash** = `Math.abs(key.hashCode()) % list.size()`

Absolute value and % (remainder when dividing by) list size ensures valid index

- `put(key, value)`
  - Add (<key, value>) to list at index hash
  - If key already there, update value
- `get(key)`
  - Return value paired with key at index hash position of list
- `containsKey(key)`
  - Check if key exists at index hash position of list

0	
1	<"hi", 5>
2	
3	
4	<"ok", 3>
5	
6	
7	

19

HashMap put/get example

- Suppose we have the <key, value> pair `<“cs”, 201>`.

Hash:

```
jshell> Math.abs("cs".hashCode()) % 8
$7 ==> 0
```

- `put(“cs”, 201)` in position 0
- `get(“cs”)` by looking up position 0, returning the value

returns 201

0	<“cs”, 201>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

20

Collisions

- Suppose now we want to put `<“fain”, 104>`.

Hash:

```
jshell> Math.abs("fain".hashCode()) % 8
$11 ==> 0
```

- `put(“fain”, 104)` in position 0
- But `<“cs”, 201>` is already stored at position 0! Call this a *collision*.

0	<“cs”, 201>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

21

Dealing with collisions: concepts

- Think of the hash table as an Array of “buckets”.
- Each bucket can store multiple <key, value> pairs.
- `put(key, value)`
  - Add to hash index bucket
  - Update value if key already in bucket
- `get(key)`
  - Loop over keys in hash index bucket
  - Return value of one that equals() key

0	<“cs”, 201> <“fain”, 104>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

22

Dealing with collisions: details

- Bucket is really another list
- Hash table is really an *array of lists* of <key, value> pairs.
- We call this technique for dealing with collisions *chaining*.

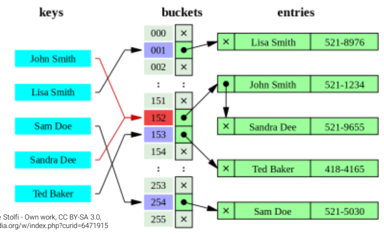


Illustration credit: By Jorge Stolfi - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6471915

23

2. HashSet and HashMap have constant time add, contains, put, get, and containsKey operations. That means that these methods... *

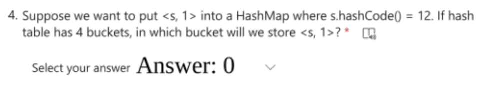
- Take the same amount of time to run
- Have the same number of operations
- [✓ Runtimes do not depend on number of elements of the Set/Map]

25

3. What is stored in each "bucket" in a HashMap? *

- A <key, value> pair
- A list of keys
- A list of values
- [✓ A list of <key, value> pairs]

26



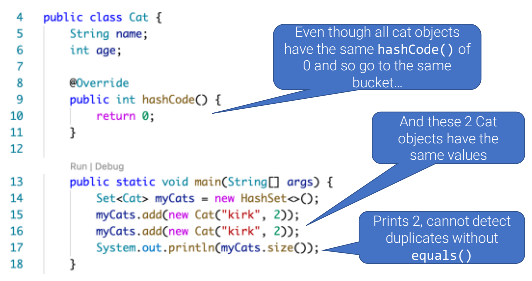
4. Suppose we want to put <s, 1> into a HashMap where s.hashCode() = 12. If hash table has 4 buckets, in which bucket will we store <s, 1>? *

Select your answer Answer: 0

Where does `equals()` come in?

- If multiple <key, value> pairs in same bucket, need to know which to get() or update on a put() call.
- Always the pair where the key in the bucket `equals()` the key we `put()` or `get()`.
- Need `equals()` to work correctly for the key type
  - String keys? Integer? Already implemented for you.
  - Storing objects of a class *you write*? Need to override and implement `equals()`.

What happens without equals()? Hashing cats



```
public class Cat {
    String name;
    int age;

    @Override
    public int hashCode() {
        return 0;
    }

    Run | Debug
    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats.size());
    }
```

Even though all cat objects have the same `hashCode()` of 0 and so go to the same bucket…

And these 2 Cat objects have the same values

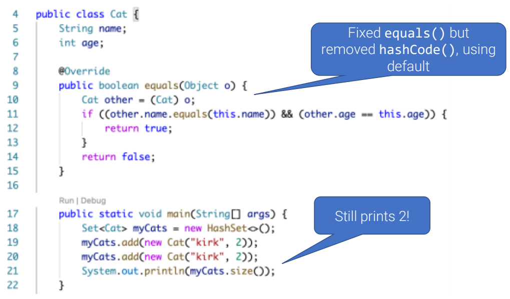
Prints 2, cannot detect duplicates without `equals()`

hashCode Correctness

- Need **`hashCode()`** to work correctly for the key type.
  - String keys? Already implemented for you.
  - Storing objects of classes *you write*? Need to override and implement **`hashCode()`**.
- What makes a **`hashCode()`** “correct” (not necessarily efficient)?
  - Any two objects that are **`equals()`** should have the same **`hashCode()`**.

30

What happens without hashCode()? Hashing more cats



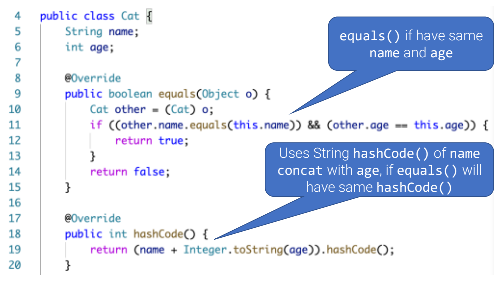
```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name.equals(this.name)) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    Run | Debug
    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats.size());
    }
```

Fixed **`equals()`** but removed **`hashCode()`**, using default

Still prints 2!

31

Cat with `equals()` and `hashCode()`



```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name.equals(this.name)) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
```

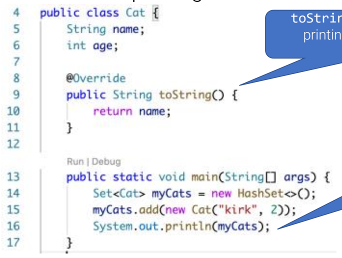
`equals()` if have same **`name`** and **`age`**

Uses String **`hashCode()`** of **`name`** concat with **`age`**, if **`equals()`** will have same **`hashCode()`**

32

Aside: `toString()`

Don't need for hashing, but **`toString()`** method allows "nice" printing.



```
public class Cat {
    String name;
    int age;

    @Override
    public String toString() {
        return name;
    }

    Run | Debug
    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats);
    }
```

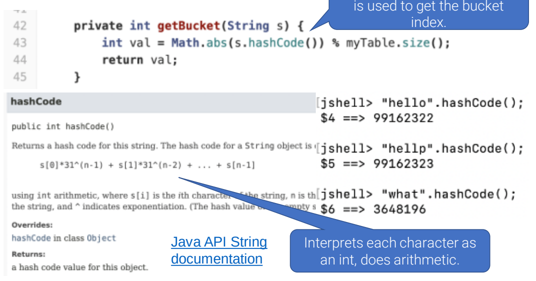
`toString()` method used for printing, including inside a Collection

Prints **`kirk`** instead of **`Cat@...`**

33

What is the String hashCode()?

Remember how **`hashCode()`** is used to get the bucket index.



```
    private int getBucket(String s) {
        int val = Math.abs(s.hashCode()) % myTable.size();
        return val;
    }
```

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is

```
    s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the ith character of the string, `n` is th the string, and ^ indicates exponentiation. (The hash value empty s

Overrides:

`hashCode` in class `Object`

Returns:

a hash code value for this object.

```
jshell> "hello".hashCode();
$4 ==> 99162322

jshell> "hellp".hashCode();
$5 ==> 99162323

jshell> "what".hashCode();
$6 ==> 3648196
```

Java API String documentation

Interprets each character as an int, does arithmetic.

34

Revisiting Hashing Efficiency

- Runtime of **`get()`**, **`put()`**, and **`containsKey()`**

  = Time to get the hash

  Constant, does not depend on number of pairs in Map

  + Time to search "bucket", calling .equals() on everything in the bucket

  Depends on number of pairs per bucket

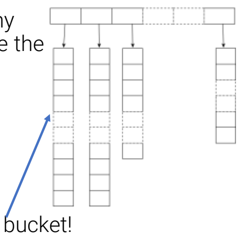
⇒ HashMaps are faster with more buckets

35

“correct” but inefficient `hashCode()`

Correctness requirement: Any `.equals()` keys should have the same `hashCode()`.

```java
@Override
public int hashCode() {
    return 0;
}
```



Still satisfies, but not good…

Stores everything in the first bucket!
No more efficient than a list!

36

Correct and efficient `hashCode()`

From the Java 17 API documentation:

- Correctness: “If two objects are equal…hashCode…must produce the same integer result.”
- Efficiency: “…producing distinct integer results for unequal objects may improve the performance of hash tables.”
- String hashCode() satisfies both

```
jshell> "hello".hashCode();
$4 ==> 99162322

jshell> "hellp".hashCode();
$5 ==> 99162323

jshell> "what".hashCode();
$6 ==> 3648196
```

37

Cat `hashCode()` revisited

```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name == this.name) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
```

`equals()` if have same `name` and `age`

If `equals()` will have same `hashCode()`

If unequal? Unlikely (but possible!) to have the same `hashCode()`.

38

Simple uniform hashing assumption (SUHA)

- Suppose we hash N pairs to M buckets.
- ***Simple uniform hashing assumption***:
  Any element (i.e., key for HashMap, value for HashSet) is **equally likely** to hash into any bucket, **independently** of where any other element hashes to. [CLRS]
- Probability any two unequal elements hash into the same bucket: 1/M
  - Spread of pairs to buckets **looks random** (but is not).
  - Ways to design such hash functions, not today
  - We make this assumption to analyze efficiency in theory, can verify runtime performance in practice

39

Implications of SUHA

- Expected number of pairs per bucket under SUHA? N/M [N pairs, M buckets].
- Stronger statements are true: Very high probability that any bucket has approximately N/M pairs.
- Runtime implication?
  - Time to get the hash

    Constant, does not depend on N or M.
  - Time to search over the hash index "bucket"
    - Calling .equals() on everything in the bucket

      Expect ~ N/M pairs to search

40