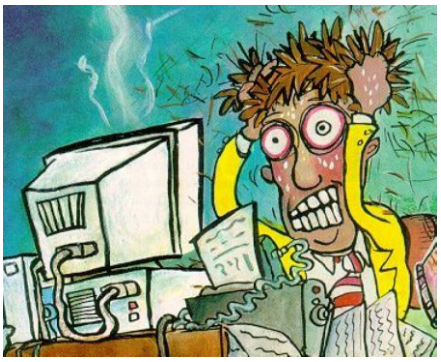
CSE 143

Computer Programming II

Sets and Maps



Outline

Alice in Wonderland

Count the Number of Distinct Words in a Text

Write a program that counts the number of unique words in a large text file (say, "Alice in Wonderland"). The program should:

- Store the words in a collection and report the number of unique words in the text file.
- Allow the user to search it to see whether various words appear in the text file.

What collection is appropriate for this problem?

We could use an `ArrayList`...

We'd really like a data structure that **takes care of duplicates for us**.

What is a Set?

Definition (Set)

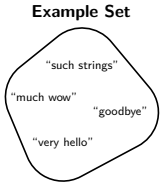
A **set** is an **unordered** collection of **unique** values. You can do the following with a set:

- Add **element** to the set
- Remove **element** from the set
- Is **element** in the set?

How To Think About Sets

Think of a set as a bag with objects in it. You're allowed to pull things out of the bag, but someone might shake the bag and re-order the items.

Example Set



Is "goodbye" in the set? **true**
Is "doge" in the set? **false**

`Set` Implementations

`Set` is an **interface** in `java.util`; implementations of that interface are:

`HashSet`

- $\mathcal{O}(1)$ for all operations.
- **Does not** maintain a useful ordering

`TreeSet`

- $\mathcal{O}(\log(n))$ for all operations
- **Does** maintain the elements in **sorted order**

Set Reference 4

Constructors

`new HashSet<E>()`	Creates a new `HashSet` of type `E` that initially has no elements
`new HashSet<E>(collection)`	Creates a new `HashSet` of type `E` that initially has all the elements in **collection**
`new TreeSet<E>()`	Creates a new `TreeSet` of type `E` that initially has no elements
`new TreeSet<E>(collection)`	Creates a new `TreeSet` of type `E` that initially has all the elements in **collection**

Methods

`add(val)`	Adds **val** to the set
`contains(val)`	Returns true if **val** is a member of the set
`remove(val)`	Removes **val** from the set
`clear()`	Removes all elements from the set
`size()`	Returns the number of elements in the set
`isEmpty()`	Returns `true` whenever the set contains no elements
`toString()`	Returns a string representation of the set such as `[3, 42, -7, 15]`


Set Reference

Looping Through Sets 5

How can we list all the elements of a set?

- We can't do a normal `for` loop, because **there are no indexes**
- We also don't know what is actually **in** the set...

Solution

The solution is a new type of loop called the **`foreach` loop**.

```
1  Set<Integer> set = new HashSet<Integer>();
2  set.add(5);
3  set.add(5);
4  set.add(5);
5  set.add(10);
6  set.add(12);
7  for (int i : set) {
8     System.out.println(i);
9  }
10 // The set remains unchanged.
```

OUTPUT

```
>> 10
>> 5
>> 12
```

`foreach` Loops 6

In general, `foreach` loops look like the following:

```
1 for (type var : collection) {
2    // do something with var
3 }
```

You can use them for many other collections like `Lists`.
You are **not allowed** to use them for `Stacks` or `Queues`.

Another Example of `foreach` Loops

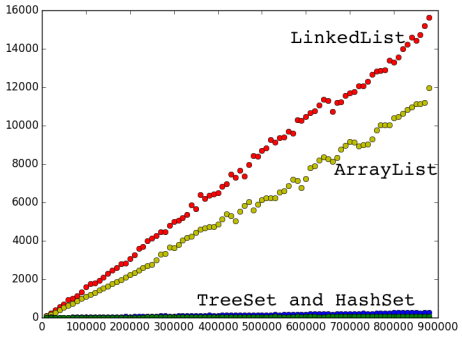
```
List<String> list = new ArrayList<String>();
list.add("a");
list.add("a");
list.add("b");
list.add("d");
String everything = "";
for (String s : list) {
   everything += s;
}
System.out.println(everything);
```

OUTPUT

```
>> aabd
```

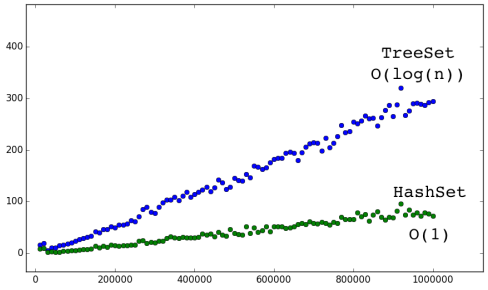
Data Structure Performance 7

The following is the performance of various data structures at removing duplicates from a large dictionary of words.



Data Structure Performance, Part 2 8

Note that despite it looking like `HashSet` and `TreeSet` have the same runtime on the previous slide, they do not.



Alice in Wonderland, Take 2 9

Count the Number of **Occurrences** of Each Word in a Text

Write a program that counts the number of unique words in a large text file (say, "Alice in Wonderland"). The program should:

- Allow the user to type a word and report how many times that word appeared in the book.
- Report all words that appeared in the book at least 500 times, in alphabetical order.

What collection is appropriate for this problem?

We could use something **sort of like** `LetterInventory`, but we don't know what the words are in advance...

We'd really like a data structure that **relates tallies with words**.

What is a Map? 10

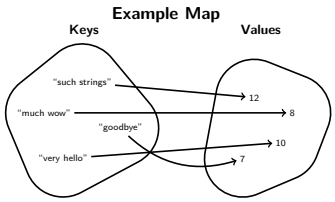
Definition (Map)

A **map** is a data structure that **relates keys** and **values**. You can do the following with a map:

- Ask what **value** a particular **key** maps to.
- Change what **value** a particular **key** maps to.
- Remove whatever the relation is for a given **key**.

How To Think About Maps

- Maps are a lot like functions you've seen in math: $f(x) = x^2$ maps 0 to 0, 2 to 4, ...
- Your **keys** are identifiers for values. Ex: social security numbers (maps SSN → person).
- Safe-deposit boxes are another useful analogy. You get a *literal* key to access your belongings. If you know what the key is, you can always get whatever you're keeping safe.



How many characters is "much wow"? 8
What does "goodbye" map to? 7
What is the value for "such strings"? 12

Map Implementations 11

`Map` is an **interface** in `java.util`; implementations of that interface are:

`HashMap`

- $\mathcal{O}(1)$ for all operations.
- **Does not** maintain a useful ordering of anything

`TreeMap`

- $\mathcal{O}(\log(n))$ for all operations
- **Does** maintain the **keys** in **sorted order**

`Map` Constructors & Type Parameters 12

Creating A Map

To create a map, you must specify **two** types:

- What type are the keys?
- What type are the values?

They **can** be the same, but they aren't always.

Constructors

`new HashMap<K,V>()`	Creates a new `HashMap` with keys of type `K` and values of type `V` that initially has no elements
`new TreeMap<K,V>()`	Creates a new `TreeMap` with keys of type `K` and values of type `V` that initially has no elements

`Map` Reference 13

`put(`**key**`,`**val**`)`	Adds a mapping from **key** to **val**; if **key** already maps to a value, that mapping is replaced with **val**
`get(`**key**`)`	Returns the value mapped to by the given **key** or `null` if there is no such mapping in the map
`containsKey(`**key**`)`	Returns true the map contains a mapping for **key**
`remove(`**key**`)`	Removes any existing mapping for **key** from the map
`clear()`	Removes all key/value pairs from the map
`size()`	Returns the number of key/value pairs in the map
`isEmpty()`	Returns `true` whenever the map contains no mappings
`toString()`	Returns a string repr. of the map such as `{d=90, a=60}`
`keySet()`	Returns a set of all keys in the map
`values()`	Returns a collection of all values in the map
`putAll(`**map**`)`	Adds all key/value pairs from the given map to this map
`equals(`**map**`)`	Returns true if given **map** has the same mappings as `this`

Using A Map 14

Each map can **answer one type of question**. For example:

If the keys are phone numbers and the values are people

Then, the map can answer questions of the form:

"Who does this phone number belong to?"

```
1 Map<String,String> people = new HashMap<String,String>();
2 people.put("(206) 616-0034", "Adam's Office");
3 people.get("(206) 616-0034"); // Returns "Adam's Office"
```

The `people` map can **only go in one direction**. If we want the other direction, we need a different map:

If the keys are people and the values are phone numbers

Then, the map can answer questions of the form:

"What is this person's phone number?"

```
1 Map<String,String> phoneNumbers = new HashMap<String,String>();
2 phoneNumbers.put("Adam's Office", "(206) 616-0034");
3 phoneNumbers.get("Adam's Office"); // Returns "(206) 616-0034"
```

Using A Map 15

Earlier, we had an example where

- keys were "phrases"
- values were "# of chars in the key"

That map can answer the question:

"How many characters are in this string?"

```
1 Map<String,Integer> numChars = new HashMap<String,Integer>();
2 numChars.put("very hello", 10);
3 numChars.put("goodbye", 7);
4 numChars.put("such strings", 12);
5 numChars.put("much wow", 8);
6 numChars.get("much wow"); // Returns 8
```

keySet

There **is no good way** to go from a **value** to its **key** using a map. But we can go from **each key** to the values:

```
1  Map<String, Double> ages = new TreeMap<String, Double>();
2  // These are all according to the internet...a very reliable source!
3  ages.put("Bigfoot", 100);
4  ages.put("Loch Ness Monster", 3.50);
5  ages.put("Chupacabra", 20); // ages.keySet() returns Set<String>
6  ages.put("Yeti", 40000);
7  for (String cryptid : ages.keySet()) {
8     double age = ages.get(cryptid);
9     System.out.println(cryptids + " -> " + age);
10 }
```

OUTPUT

```
>> Chupacabra -> 20
>> Loch Ness Monster -> 1500
>> Bigfoot -> 100
>> Yeti -> 40000
```

values

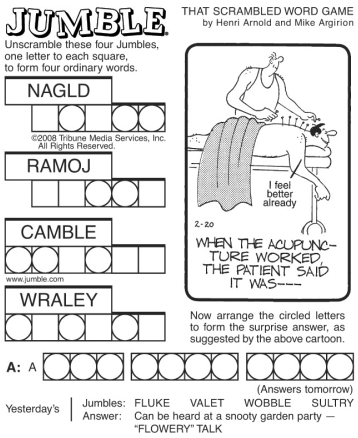
You **can** get a collection of all the values:

```
1  Map<String, Double> ages = new TreeMap<String, Double>();
2  // These are all according to the internet...a very reliable source!
3  ages.put("Bigfoot", 100);
4  ages.put("Loch Ness Monster", 3.50);
5  ages.put("Chupacabra", 20); // ages.keySet() returns Set<String>
6  ages.put("Yeti", 40000);
7
8  for (int age : ages.values()) {
9     System.out.println("One of the cryptids is aged " + age);
10 }
```

OUTPUT

```
>> One of the cryptids is aged 1500
>> One of the cryptids is aged 40000
>> One of the cryptids is aged 20
>> One of the cryptids is aged 100
```

Word Jumbles



Some Set/Map Tips!

- Sets and Maps are two more collections each with their own places
- Sets are for storing data **uniquely**
- Maps are for storing **relationships** between data; they only **work in one direction**
- `foreach` loops are a great tool for looping through collections
- You should know the syntax for `foreach` loops and that `Hash` and `Tree` are types of sets and maps