CS 2

Introduction to Programming Methods

Java Collections

Abstract Data Types (ADT)

Abstract Data Type

An **abstract data type** is a description of what a collection of data **can do**. We usually specify these with **interfaces**.

`List` ADT

In Java, a **List** can `add`, `remove`, `size`, `get`, `set`.

`List` Implementations

An **ArrayList** is a particular type of List. Because it is a list, we promise it can do everything a List can. A **LinkedList** is another type of List.

Even though we don't know how it works, we know it can do everything a List can, **because it's a List**.

Using the `List` ADT

This is INVALID CODE

```
List<String> list = new List<String>(); // BAD : WON'T COMPILE
```

`List` is a description of methods. It doesn't specify **how they work**.

This Code Is Redundant

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}

LinkedList<Integer> list = new LinkedList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}
```

We can't condense it any more when written this way, because `ArrayList` and `LinkedList` are totally different things.

NOT Using the List ADT

Instead, we can use the List interface and swap out different implementations of lists:

This Uses Interfaces Correctly!

```
List<Integer> list = new ArrayList<Integer>();
                // = new LinkedList<Integer>();
                // We can choose which implementation
                // And the code below will work the
                // same way for both of them!
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}
```

The other benefit is that the code doesn't change based on which implementation we (or a client!) want to use!

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?

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)

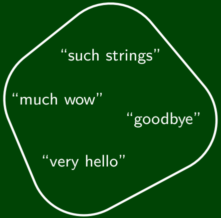
A **set** is an **unordered** collection of **unique** values. You can do the following with a set:

- 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:

`TreeSet`

- **Really fast**
- **Does** maintain the elements in **sorted order**

`HashSet`

- **REALLY REALLY fast**
- **Does not** maintain a useful ordering

`Set` Reference

Constructors

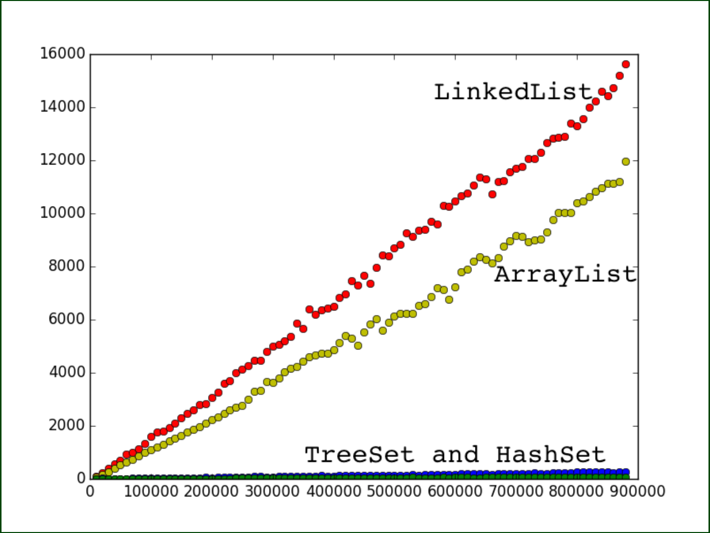
`new HashSet<E>()`	Creates a new `HashSet` of type `E` that initially has no elements
`new HashSet<E>(`**collection**`)`	Creates a new `HashSet` of type `E` that initially has all the elements in **collection**
`new TreeSet<E>()`	Creates a new `TreeSet` of type `E` that initially has no elements
`new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

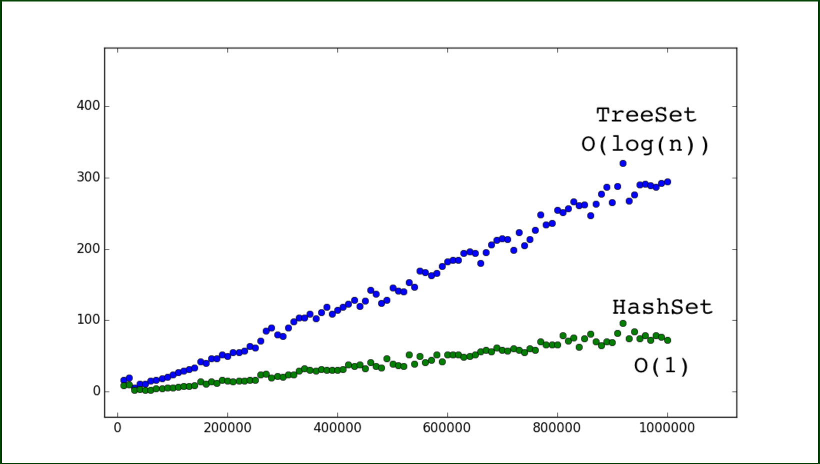
Data Structure Performance

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



Data Structure Performance, Part 2

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

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?

Alice in Wonderland, Take 2

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?

Definition (Map)

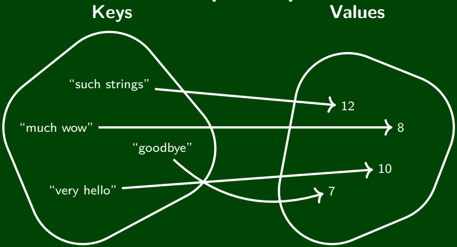
A **map** is a data structure that **relates keys** and **values**. You can do the following with a map:

- 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.

Example Map



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

Map Implementations

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

`TreeMap`

- Really fast for all operations.
- **Does** maintain the **keys** in **sorted order**

`HashMap`

- REALLY REALLY fast for all operations.
- **Does not** maintain a useful ordering of anything

`Map` Constructors & Type Parameters

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

`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

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?"

```java
Map<String,String> people = new HashMap<String,String>();
people.put("(206) 616−0034", "Adam’s Office");
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?"

```java
Map<String,String> phoneNumbers = new HashMap<String,String>();
phoneNumbers.put("Adam’s Office", "(206) 616−0034");
phoneNumbers.get("Adam’s Office"); // Returns "(206) 616−0034"
```

Using A Map

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?"

```java
Map<String,Integer> numChars = new HashMap<String,Integer>();
numChars.put("very hello", 10);
numChars.put("goodbye", 7);
numChars.put("such strings", 12);
numChars.put("much wow", 8);
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:

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

OUTPUT

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

values

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

```
Map<String, Double> ages = new TreeMap<String, Double>();
// These are all according to the internet...a very reliable source!
ages.put("Bigfoot", 100);
ages.put("Loch Ness Monster", 3.50);
ages.put("Chupacabra", 20); // ages.keySet() returns Set<String>
ages.put("Yeti", 40000);

for (int age : ages.values()) {
   System.out.println("One of the cryptids is aged " + age);
}
```

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
```

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