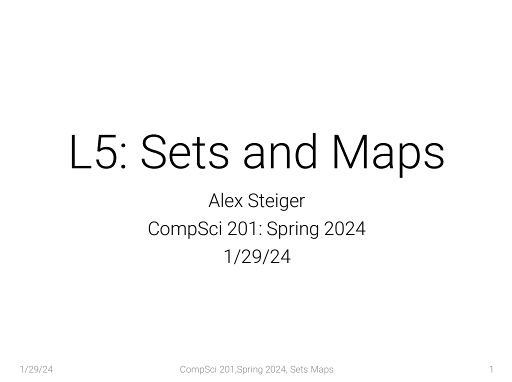
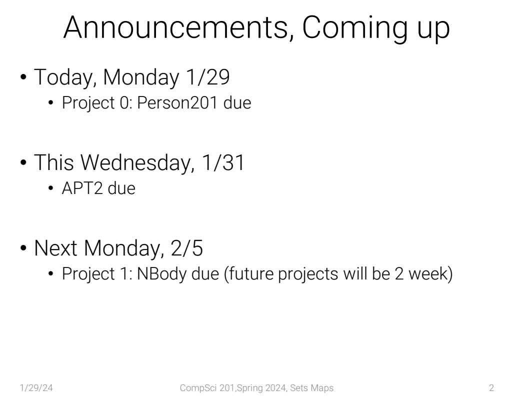


L5: Sets and Maps

Alex Steiger
CompSci 201: Spring 2024
1/29/24

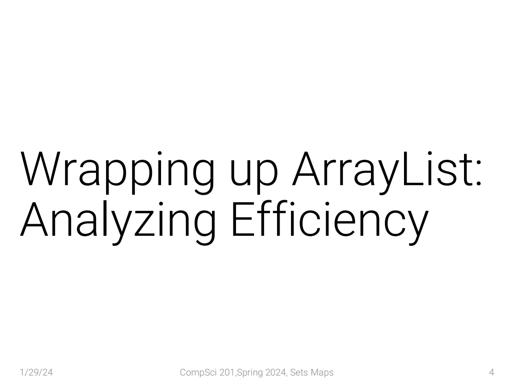
1



Announcements, Coming up

- Today, Monday 1/29
  - Project 0: Person201 due
- This Wednesday, 1/31
  - APT2 due
- Next Monday, 2/5
  - Project 1: NBody due (future projects will be 2 week)

2



Wrapping up ArrayList: Analyzing Efficiency

4

Algorithmic tradeoffs depend on the implementation

Often, we are interested in how the **efficiency** of operations on data structures depends on **scale**. For an **ArrayList** with N values how efficient is…

- `get()`. Direct lookup in an Array. "Constant time" – does not depend on size of the list.
- `contains()`. Loops through Array calling `.equals()` at each element. Takes longer as list grows.
- `size()`. Returns value of an instance variable tracking size, does not depend on size of the list.
- `add()`. Depends.

How efficient is `ArrayList` `add`?

For an **ArrayList** with N values, 2 cases:

1. Space left – One Array assignment statement, *constant time*, does not depend on list size.
2. No space left – Copy entire list! Takes N array assignments!

How often are we in the second slow case? Depends on *how much we increase the Array size by in case 2.*

Code Recap + WOTO

Live Coding

ArrayList Growth

Starting with a length 1 Array, if you add N elements one at a time and (when full) create a new Array that…

Is twice as large (geometric growth)

- Must copy at sizes:
  - 1, 2, 4, 8, 16, 32, …
- Total values copied looks like:
  - 1+2+4+8+…+(N/4)+(N/2)

Has 1 more position (arithmetic growth)

- Must copy at sizes:
  - 1, 2, 3, 4, …
- Total values copied looks like:
  - 1+2+3+…+(N-2)+(N-1)

Algebra to our rescue!

ArrayList Growth and Algebra

Geometric growth

$$1 + 2 + 4 + \cdots + (N/2)$$

$$= \sum_{i=0}^{\log_2 N-1} 2^i$$

$$= N - 1$$

Geometric series formula: $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$

Arithmetic growth

$$1 + 2 + 3 + \cdots + (N-1)$$

$$= \sum_{i=1}^{N-1} i$$

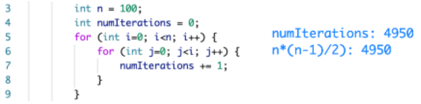
$$= N(N-1)/2$$

Arithmetic series formula: $\sum_{i=1}^{n} a_i = \left(\frac{n}{2}\right)(a_1 + a_n)$

Math and Expectations in 201

- *Do not* expect you to formally derive closed form expressions / give proofs.
- *Do* expect you to recognize:
  - $1 + 2 + 4 + \cdots + N$ is *linear*, grows like $\approx N$.
  - $1 + 2 + 3 + \cdots + N$ is *quadratic*, grows like $\approx N^2$.
- Patterns like these show up again and again!

Will make "like" more formal with asymptotic analysis



```java
int n = 100;
int numIterations = 0;
for (int i=0; i<n; i++) {
    for (int j=0; j<i; j++) {
        numIterations += 1;
    }
}
```

```
numIterations: 4950
n*(n-1)/2): 4950
```

Experiment to verify hypothesis

Live Coding

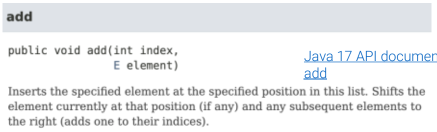
11

`ArrayList` `add` (to end) is (amortized) efficient

- According to the Java 17 API documentation: "The add operation runs in *amortized constant time…*"
  - What does that mean?
- With geometric growth (e.g., grow array by doubling size):
  - Only need a *linear* number of copies (i.e., $\propto N$ copies) to **`add`** N elements to `ArrayList`.
  - The *average* number of copies per add is thus $\propto \frac{N}{N} = 1$, a *constant* that does not depend on N.

12

`ArrayList` add to the front is not efficient



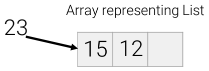
add

```
public void add(int index,
                E element)
```

Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

[Java 17 API documentation of add]

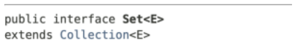
Always requires shifting the entire **`Array`**, even if there is space available.



13

Sets

Set ADT Review



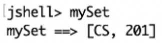
```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements.

Java API documentation

- Stores UNIQUE elements
- Check if element in Set (using `.contains()`)
- Add element to set (using `.add()`)
    - Returns `false` if already there
- Remove element (with `.remove()`)
- Not guaranteed to store them in the order added

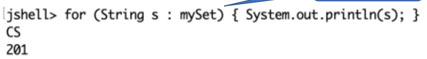
Set FAQs



```
jshell> mySet
mySet ==> [CS, 201]
```

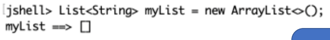
1. How do I loop over a Set?

Enhanced for loop

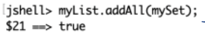


```
jshell> for (String s : mySet) { System.out.println(s); }
CS
201
```

2. How do I convert between lists and sets?

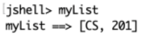


```
jshell> List<String> myList = new ArrayList<>();
myList ==> []
```



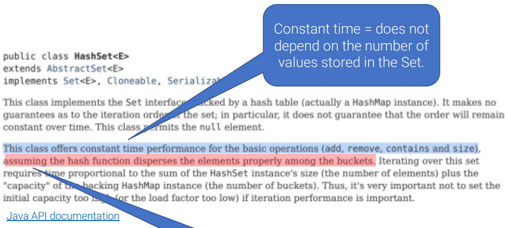
```
jshell> myList.addAll(mySet);
$21 ==> true
```

addAll() method convenient, same as looping and adding one at a time



```
jshell> myList
myList ==> [CS, 201]
```

HashSet implementation of Set is very efficient



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

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits 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 time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

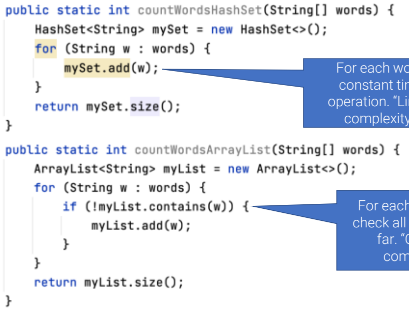
Java API documentation

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

Under assumptions we will discuss next time

17

Count Unique Words?



```
public static int countWordsHashSet(String[] words) {
    HashSet<String> mySet = new HashSet<>();
    for (String w : words) {
        mySet.add(w);
    }
    return mySet.size();
}
public static int countWordsArrayList(String[] words) {
    ArrayList<String> myList = new ArrayList<>();
    for (String w : words) {
        if (!myList.contains(w)) {
            myList.add(w);
        }
    }
    return myList.size();
}
```

For each word, constant time operation. "Linear complexity."

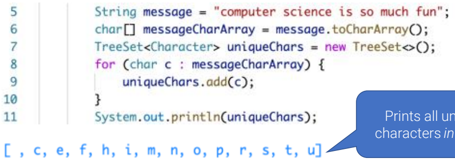
For each word, must check all the words so far. "Quadratic complexity."

18

`TreeSet` stores sorted

Two important implementations of Set interface:

- `HashSet` – Very efficient `add`, `contains`
- `TreeSet` – Nearly as efficient, keeps values sorted by their "*natural ordering*"



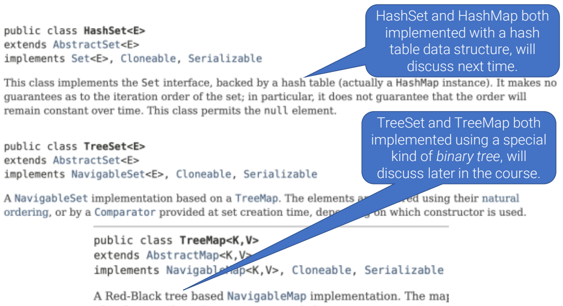
```
        String message = "computer science is so much fun";
        char[] messageCharArray = message.toCharArray();
        TreeSet<Character> uniqueChars = new TreeSet<>();
        for (char c : messageCharArray) {
            uniqueChars.add(c);
        }
        System.out.println(uniqueChars);
```

```
[ , c, e, f, h, i, m, n, o, p, r, s, t, u]
```

Prints all unique characters *in order*.

19

HashSet and TreeSet Implementations



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

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

```
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based NavigableMap implementation. The map

HashSet and HashMap both implemented with a hash table data structure, will discuss next time.

TreeSet and TreeMap both implemented using a special kind of *binary tree*, will discuss later in the course.

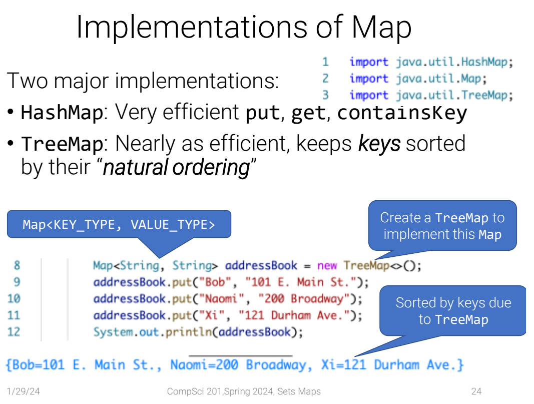
Maps

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



Implementations of Map

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

Two major implementations:

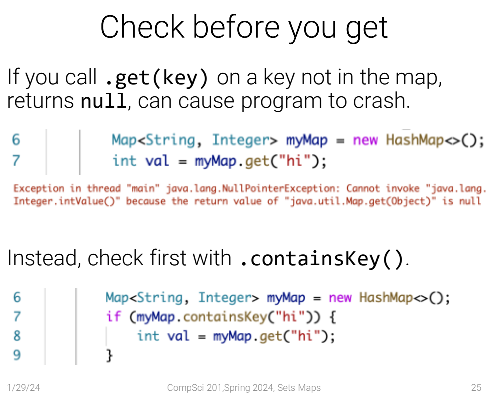
- `HashMap`: Very efficient `put`, `get`, `containsKey`
- `TreeMap`: Nearly as efficient, keeps ***keys*** sorted by their "*natural ordering*"

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);
```

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

24



Check before you get

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

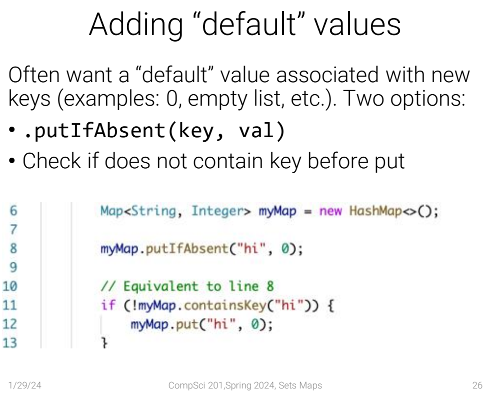
```java
Map<String, Integer> myMap = new HashMap<>();
int val = myMap.get("hi");
```

```
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");
}
```

25



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

```java
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  }
```

26

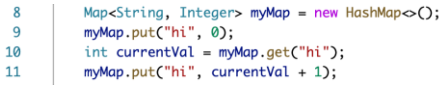
Updating maps

Immutable values:

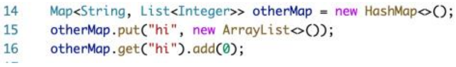
- `.get()` returns a *copy of the value.*
- Must use `.put()` again to update.

Mutable values (e.g. collections)

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



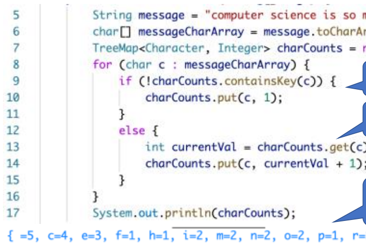
```java
Map<String, Integer> myMap = new HashMap<>();
myMap.put("hi", 0);
int currentVal = myMap.get("hi");
myMap.put("hi", currentVal + 1);
```



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

Counting with a `Map`

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



```java
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);
```

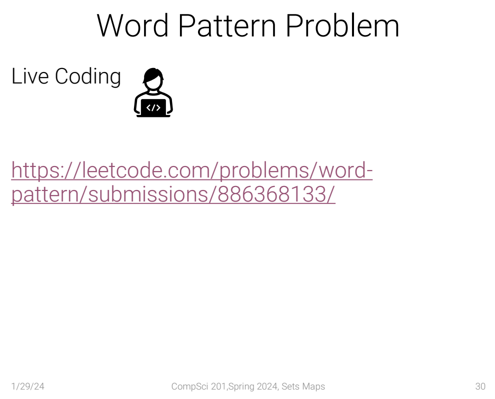
```
{ =5, c=4, e=3, f=1, h=1, i=2, m=2, n=2, o=2, p=1, r=1, s=3, t=1, u=3}
```

Check if we have *not* seen c yet

Else get current value and increase

Comes in order because using `TreeMap`

Problem-Solving with Sets and Maps



Word Pattern Problem

Live Coding

https://leetcode.com/problems/word-pattern/submissions/886368133/

30