L5: Sets and Maps
Alex Steiger
CompSci 201: Spring 2024
1/29/24

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)

Wrapping up ArrayList:
Analyzing Efficiency
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!

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!
Experiment to verify hypothesis

Live Coding 🎨

**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 \( \frac{N}{N} = 1 \), a constant that does not depend on \( N \).

**ArrayList add to the front is not efficient**

```java
public void add(Integer index, Integer element)
```

Java 17 API documentation of `add`

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 add one to their indices.

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

Set ADT Review

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

Java API documentation

Set FAQs

1. How do I loop over a Set?
   ```java
   $jshell$ mySet
   mySet => [C, 121]
   $jshell$ for (String s : mySet) { System.out.println(s); } // C
   ```

2. How do I convert between lists and sets?
   ```java
   $jshell$ List<String> myList = new ArrayList<>();
   $jshell$ myList.addAll(mySet);
   $jshell$ $21 == true
   $jshell$ myList.add(1, mySet);
   ```

Enhanced for loop

\texttt{.addAll()} method convenient, same as looping and adding one at a time
HashSet implementation of Set is very efficient

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

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

This class offers constant time performance for the basic operations (add, remove, contains and size).
Ensuring the hash function disperses the elements properly among the buckets. Iterating over this set
requires O(n) time proportional to the sum of the HashSet instances size (the number of elements) plus the
"capacity" (max number of elements the HashSet instance can hold before resizing). Thus, it's very important not to set
the initial capacity too high, as this may result in bad memory utilization. A decision performance is important.

 HashSet documentation
```

Under assumptions we will discuss next time

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."

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);
[ i, c, h, n, l, o, p, r, s, t, u ]
```

Prints all unique characters in order.
HashSet and TreeSet Implementations

HashSet and HashSet 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.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>101 E. Main St.</td>
</tr>
<tr>
<td>Naomi</td>
<td>200 Broadway</td>
</tr>
<tr>
<td>Stavros</td>
<td>121 Durham Ave.</td>
</tr>
</tbody>
</table>

• 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

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

```
Maps<KEY_TYPE, VALUE_TYPE>
```

```
Map<String, String> addressBook = new TreeMap<>();
addressBook.put("Bob", "101 E. Main St.");
addressBook.put("Sam", "200 Broadway");
addressBook.put("Bob", "211 Burhan Ave.");
System.out.println(addressBook);
```

Sorted by keys due to TreeMap

Check before you get

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

```
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()`.

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

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

```
Map<String, Integer> myMap = new HashMap<>();
myMap.putIfAbsent("hi", 0);
```

```
// Equivalent to line 8
if (!myMap.containsKey("hi")) {
    myMap.put("hi", 0);
}
```
### Updating maps

<table>
<thead>
<tr>
<th>Immutable values:</th>
<th>Mutable values (e.g. collections)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>.get()</code> returns a copy of the value.</td>
<td>• <code>.get()</code> returns reference to collection.</td>
</tr>
<tr>
<td>• Must use <code>.put()</code> again to update.</td>
<td>• Update the collection directly.</td>
</tr>
</tbody>
</table>

```java
8  Map<String, Integer> myMap = new HashMap<>();
9  myMap.put("hi", 0);
10 int currentVal = myMap.get("hi");
11  myMap.put("hi", currentVal + 1);
12  Map<String, List<Integer>> otherMap = new HashMap<>();
13  otherMap.put("hi", new ArrayList<>());
14  otherMap.get("hi").add(0);
```

---

### Counting with a Map

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

```java
5  String message = "computer science is so much fun";
6  char[] messageChars = message.toCharArray();
7  TreeMap<Character, Integer> charCounts = new TreeMap<>();
8  for (char c : messageChars) {
9    if (!charCounts.containsKey(c)) {
10      charCounts.put(c, 1);
11    } else {
12      int setCurrentVal = charCounts.get(c);
13      charCounts.put(c, setCurrentVal + 1);
14    }
15  }
16  System.out.println(charCounts);
```

<table>
<thead>
<tr>
<th>if we have not seen c yet</th>
<th>Else get current value and increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comes in order because using TreeMap</td>
<td></td>
</tr>
</tbody>
</table>

---

### Problem-Solving with Sets and Maps
Word Pattern Problem

Live Coding

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