CSE 2123: Collections: Maps

Jeremy Morris
Collections - Map

- A map is another kind of collection
  - Different from previously discussed examples
  - Lists, Queues, Stacks are all linear collections
    - Some meaning to the underlying ordering of the data
    - Data elements at particular indices, or at the head or tail or interior of the list, etc.
  - Sets are unordered collections
    - Data is unordered
    - We only care whether things are in the Set or not in the Set, not position
- Maps are relational collections
Collections - Map

- Maps are *relational* collections
  - What does this mean?
- Maps create an association between a *key* and a *value*
  - The *value* is just like the data element in a list
  - The *key* is like an index in a list – but it can be anything, not just a number
  - Maps establish a relationship between a key and a particular value
    - Keys must be unique, values can repeat
Maps are sometimes also known as *dictionaries*

- Useful analogy – real world dictionary
- The *key* in a dictionary is the word you are looking up
- The *value* in a dictionary is the definition of the word
- Dictionary is organized by key – by the words you might want to look up.
Collections - Map

- A map of Student objects
  - We might want to be able to find grades according to a student ID number
    - The student ID number is the key
    - The Student object is the value

- A map of Car objects
  - Find car objects based on license plate number
    - The license plate number is the key
    - The Car object is the value
Maps rely on a few standard methods:

- Put an object into the Map with a key:
  \[ E \text{ put}(K \text{ key}, E \text{ value}) \]

- Retrieve the object associated with a key
  \[ E \text{ get}(K \text{ key}) \]

- Remove an object from the Map based on a key
  \[ E \text{ remove}(K \text{ key}) \]
Map methods (cont):

- Get the number of mappings:
  ```java
  int size()
  ```

- Check to see if the Map is empty
  ```java
  boolean isEmpty()
  ```

- Clear the map of all elements
  ```java
  void clear()
  ```
Collections - Map

- Java implements different kinds of Maps
  - Each is more efficient under different circumstances
  - We will talk about HashMaps and TreeMaps
    - Sound familiar?
  - HashMaps use a hash function to associate keys and values
  - TreeMap stores values in a binary tree ordered by their keys
HashMap of student GPAs by name

- Student name used as the key
- Double objects (holding GPA) used as the value

What happens if two students have the same name?
Collections - Map (HashMap)

- **HashMap of StudentGrade objects**
  - StudentIDs used as the *key*
  - StudentGrade objects used as the *value*
    - Using StudentIDs ensures we have a unique value

```
HashMap<String, Student>
```

```
“123456789” → StudentGrade (contains info for “Bob Marley”)  
“234567890” → StudentGrade (contains info for “Joe Cool”)  
“345678901” → StudentGrade (contains info for “Charles Brown”)  
```
Declaring a HashMap requires that we know a types for the *key* and for the *value*

- A HashMap with Strings as keys and Doubles as values:
  ```java
  Map<String,Double> myMap = new HashMap<String,Double>();
  ```

- A HashMap with Strings as keys and Student objects as values:
  ```java
  Map<String,Student> myMap = new HashMap<String,Student>();
  ```
public static void main(String[] args) {
    Map<String, Double> myMap = new HashMap<String, Double>();

    myMap.put("Marley, Bob", 3.0);
    myMap.put("Cool, Joe", 2.7);
    myMap.put("Brown, Charles", 3.5);
    System.out.println(myMap);

    System.out.println(myMap.get("Cool, Joe");
    System.out.println(myMap.get("Brown, Charles");
    System.out.println(myMap.get("Marley, Bob");
}
HashMap of names, GPAs

- Key/Value pairs (Entries) are stored in “buckets”
  - Each bucket has an associated index value
  - The key’s hash function tells us which bucket to use
  - When we need to look up a value, just apply the hash function to the key again
    - Result tells us which bucket to look in
What happens if two keys have the same hash function result?

- As with the HashSet, this is known as a *collision*
  - In these cases, the HashMap first checks to see if the keys really are equal (using the key’s equals() method).
  - If they are not equal, then it must maintain both Entries at the same index
What happens if two elements have the same hash function result?

- This is known as a collision
  - In these cases, the HashMap first checks to see if they really are equal (using the objects equals() method).
  - If they are not equal, then it must maintain both Entries at the same index
  - One method: use a LinkedList of values for the buckets
HashMaps – User Defined Classes

- Note that if you’re using your own classes as the *key* in a HashMap, you *must* have a well-defined hashCode method
  - The hashCode method is never used for objects that are stored as *values* in a HashMap
Collections – Map (TreeMap)

TreeMap<String, Double>

- TreeMap of student GPAs by name
  - Student name used as the key
  - Stored in a binary tree (red-black tree) according to the key
  - Double objects (holding GPA) used as the value

- “Brown, Charles” → 3.5
- “Cool, Joe” → 2.7
- “Marley, Bob” → 3.0
Collections - TreeMap

Tree

“Cool, Joe”, 2.7

“Brown, Ch...”, 3.5

“Marley, Bob”, 3.0

- TreeMap of names, GPAs
  - Key/Value pairs (Entries) are stored in nodes
    - The key’s compareTo method tells us where it will be stored in the tree
    - When we need to look up a value, traverse the tree by using the compareTo method
TreeMaps – User Defined Classes

- Note that if you’re using your own classes as the key in a TreeMap, you must implement the Comparable interface.
  - Objects stored as values in a TreeMap do not use compareTo – only objects that are used as keys.
Collections - Map

- Often with a Map we want to get back the keys contained in the Map
  - Common Map usage:
    - Get keys used by Map
    - Iterate over Map using keys performing some kind of processing on the values stored in the map

- Java Maps provide a method to retrieve keys
  
  \[
  \text{Set}\langle K \rangle \ \text{keySet}() \\
  \]
  - Returns a collection of keys organized as a Set
Collections – Sets and Maps

- Common Map usage:
  - Get keys used by Map
  - Iterate over Map using keys performing some kind of processing on the values stored in the map

- Common Set usage:
  - Get an iterator from the Set
  - Use the iterator to examine each item in the Set in turn

- Now we see how to implement this in Java:
  - Get keys used by Map using `keySet()`
  - Get an Iterator from the Set of keys
  - Iterate over keys to access elements in the Map
What kind of Set will be returned by keySet()?

- It depends on the type of Map you’re using
  - HashMap returns a HashSet
  - TreeMap returns a TreeSet
- But it doesn’t matter – you shouldn’t worry about the Set implementation at all
  - Just use the Set interface
HashMap vs. TreeMap

- When should we use a HashMap and when should we use a TreeMap
  - If we want to iterate over our keys in sorted order, then we should use a TreeMap
  - Otherwise, HashMap is a better choice
    - Why is this?
HashMap vs. TreeMap

- Think of the basic ‘get’ and ‘put’ operations:
  - For a HashMap, get and put can operate in \( constant \) time
    - IF you have a good hash function that distributes Entries evenly among buckets
    - Time to access a bucket is constant – perform the hash to find the bucket then access the bucket
  - For a TreeMap, get and put operate in \( log(n) \) time
    - This means that the speed that these operate is a function of the number of elements in the collection \( n \)
    - In this case, the function is the base-2 log of the number of elements
Binary Tree Access

- Suppose we have this binary tree:
  - The value of n=1
  - How many operations does it take to find the node with the value 1?
Binary Tree Access

- Suppose we have this binary tree:
  - $n = 3$
  - How many operations does it take to find the node with the value 1?
Binary Tree Access

Suppose we have this binary tree:

- n = 7
- How many operations does it take to find the node with the value 1?
Binary Tree Access

- Suppose we have this binary tree:
  - $n = 15$
  - How many operations does it take to find the node with the value 1?
Binary Tree Access

- Suppose we have this binary tree:
  - $n = 15$
  - How many operations does it take to find the node with the value 1?
  - The number of operations needed increases as we increase nodes
    - But NOT in a linear fashion
    - Logarithmic
public static void main(String[] args) {
    HashMap<String, Double> myMap = new HashMap<String, Double>();
    myMap.put("Marley, Bob", 3.0);
    myMap.put("Cool, Joe", 2.7);
    myMap.put("Brown, Charles", 3.5);
    System.out.println(myMap);
    Set<String> keys = myMap.keySet();
    Iterator<String> keyIter = keys.iterator();
    while (keyIter.hasNext()) {
        String currentKey = keyIter.next();
        System.out.print(currentKey + ": ");
        System.out.println(myMap.get(currentKey));
    }
}
In-class Practice

- Write a short program that
  - Creates 3 Student objects
  - Sets their first name, last name, and StudentId fields
    - You may use the constructor that does this
  - Stores them in a Map
    - Use StudentId as the key
  - Iterates through the Map and prints the first name, last name and StudentId for each object
Multimaps

- One useful extension to Maps is the idea of a *multimap*
  - A multimap is a map where each key can be associated with a *list* of values instead of just a single value
  - We can implement a multimap in Java by using a Map with a List (or a Set) as the value in the key/value pair
Multimap

- A MultiMap with Strings as keys and Doubles as values:
  ```java
  Map<String, List<Double>> myMap =
      new HashMap<String, List<Double>>();
  ```

- A Multimap with Strings as keys and Student objects as values:
  ```java
  Map<String, List<Student>> myMap =
      new HashMap<String, List<Student>>();
  ```
Map<String, List<Integer>> myMap =
    new HashMap<String, List<Integer>>();
String[] keys = {"Test1", "Test2", "Test3"};
int m = 1;
for (int i=0; i<keys.length; i++) {
    String key = keys[i];
    for (int j=0; j<10; j++) {
        if (myMap.containsKey(key)) {
            List<Integer> list = myMap.get(key);
            list.add(j*m);
        } else {
            List<Integer> list = new LinkedList<Integer>();
            list.add(j*m);
            myMap.put(key, list);
        }
    }
    m=m+1;
}