Alan Wu

MSCI 240 Fall 2018

Instructor: Dr. Mark Hancock

November 19, 2018

Project 2

Introduction

The report will outline and delve in the implementation of three different map structures to process and analyze data from NHL stats. Our goal here is to determine who makes the most plays from the 2012-2013 season to the 2017 to 2018 season.

Implementation

ArrayList

a) The worst case growth rate of the ArrayList function is observed whenever there are n unique players in HockeyData provided in the Excel spreadsheet. Therefore the *isThere* method (see the code snippet below) that checks whether the player is unique or not will have to iterate through the entire PlayCount ArrayList as it grows larger each time a new unique player is added.

```
public static int isThere (String data, ArrayList <PlayCount> playCount) {
   for (int i = 0; i<playCount.size(); i++) {
         if (data.equals(playCount.get(i).name)) {
               return i;
         }
   }
   return -1;
}
```

To be more specific, the worst case growth would be $\frac{n\,(n-1)}{2}$, which is $O(n^2)$. I have conducted a test to make sure that the run time is rough n square. As you can see in the chart below, when the input n double, the run time quadruple. It is evident that the growth rate is $O(n^2)$. Note that the *MAX_ENTRIES* for the test is 100000 and 200000 respectively.

n = 100000		**n = 200000**	
0.123991863	0.12347257	0.417968061	0.424121176

b) For the empirical time to count for this map given the original data size, I have utilize the StopWatch class and its methods to time the time taken to finish the count for each map (see below for the methods that I implemented)

```
Stopwatch arraylistTime = new Stopwatch(); //reset and create a new timer
      arraylistTime.reset();
      arraylistTime.start();

       arraylistTime.stop();
```

For accuracy and validity, I ran the test five times and put the times down in the table below. Note that the *MAX_ENTRIES* for the test is `4000000`.

Trial	1	2	3	4	5	
	12.806490291	13.5321052	11.2179504	12.8305076	12.9410026	**Average:** 12.6656

c) The outputs (the number of unique player as well as the top 20 players) for all three are consistent. The output below is the with the *MAX_ENTRIES* of `4000000`. Please see below for the output of all three map structures.

```
Found 1750 unique players.
The 20 players with the most frequent plays and their counts are:
1. Sidney Crosby    17294
2. Patrice Bergeron    15761
3. Jonathan Toews    15111
4. Anze Kopitar    14671
5. Claude Giroux    14296
6. Mikko Koivu    13537
7. Derek Stepan    13391
8. Nicklas Backstrom    13386
9. John Tavares    13316
10. Ryan Kesler    13108
11. Tomas Plekanec    13066
12. Ryan Getzlaf    12781
13. Ryan Johansen    12366
14. Henrik Lundqvist    12366
15. Joe Pavelski    12064
16. Braden Holtby    11950
17. Derick Brassard    11915
18. Kyle Turris    11816
19. Nazem Kadri    11541
20. Tuukka Rask    11430
```

TreeMap

a) The worst case growth rate of the TreeMap structure would be $nlon(n)$, since structure of the TreeMap is of a binary search tree. Insertion of a binary search tree would require $lon(n)$ runtime and this has be to performed for every HockeyData's input (n). This can be seen below:

```
for (HockeyData data : plays) {
         if(treeMap.containsKey(data.get(0))) {
               treeMap.put(data.get(0), treeMap.get(data.get(0))+1);//get the
player's count by searching for its key
         }
               else {
                     treeMap.put(data.get(0),1); //create a new key value
pair if the player is unique, count will be 1
               }
```

As you can see in the chart below, when the input n double, the run time double. It is evident that the growth rate is $O(nlog(n)$. Note that the *MAX_ENTRIES* for the test is 100000 and 200000 respectively.

n = 100000		**n = 200000**	
0.038189577	0.038189577	0.06650492300000001	0.06715549700000001

b) For the empirical time to count for this map given the original data size, I have utilize the StopWatch class and its methods to time the time taken to finish the count for each map (see below for the methods that I implemented). For accuracy and validity, I ran the test five times and put the times down in the table below. Note that the *MAX_ENTRIES* for the test is `4000000`.

Trial	1	2	3	4	5	
	6.521433979	6.810737460	7.140342	6.627596284	6.814529001	**Average:** 6.7829277448

c) Please see above at the ArrayList section

HashMap

a) For HashMap, the worst-case growth rate is simply n owing to the constant complexity $O(1)$ when searching in hash map. This operation will be performed for each unique player in HockeyData (every player is unique in the worst case scenario). Thus, the run time for HashMap would be $O(n)$. See below for the *for loop* that iterate through the map:

```
for (HockeyData data: plays) {
                    if(hashMap.containsKey(data.get(0))) {
                           hashMap.put(data.get(0), hashMap.get(data.get(0))+1);
                    }
                           else {
                                  hashMap.put(data.get(0),1);
                           }
                    }
```

As you can see in the chart below, when the input n double, the run time double. It is evident that the growth rate is $O(n)$. Note that the *MAX_ENTRIES* for the test is 100000 and 200000 respectively.

n = 100000		**n = 200000**	
0.029229943	0.026589351	0.05011509	0.047445325

b) For the empirical time to count for this map given the original data size, I have utilize the StopWatch class and its methods to time the time taken to finish the count for each map (see below for the methods that I implemented). For accuracy and validity, I ran the test five times and put the times down in the table below. Note that the *MAX_ENTRIES* for the test is `4000000`.

Trial	1	2	3	4	5	
	5.601837657	3.67800676	5.630306529	4.931293100	5.6952473720	**Average:** 5.1073382836

c) Please see above at the ArrayList section

Discussion

Performance

a) The performance of the unsorted ArrayList is very poor as we can see from it's steep growth rate as well as average run-time, which is 12.6656. This run time is the slowest out of the three designs. The disadvantage of having a n square design really shows when the size of the input data increases.

b) The performance of the TreeMap has the second best run time out of the three owing to it's $nlog(n)$ growth rate, averaging 6.7829277448.

c) The performance of the HashMap is the best out of all three designs, averaging 5.1073382836. Given its constant $O(1)$ run time for insertion and search, it does have an edge on TreeMap which uses $\log(n)$ binary search. The large input does not drastically increase the run time like it does to ArrayList. In short, due to the above-mentioned analysis on the growth rate and run time of all three designs, HashMap seems like the correct choice in this case.

Best Design Choice

a) After reading the manual, I was left with choosing between two methods to find the top 20 players. My first instinct is to sort the entire list by Count value in the PlayCount object and thus I can pick the largest 20. I implemented a `compareTo` method in the PlayCount class and used `Collection.sort` first to sort the entire list and then a `for loop` to get the last 20 elements. Using the Collection class to sort the entire list will usually grant a stable run time of $O(nlog(n))$ where merge sort and quick sort are common sorting algorithm implemented by compiler to sort comparable class objects. Please see below for the implementation of Collection.sort:

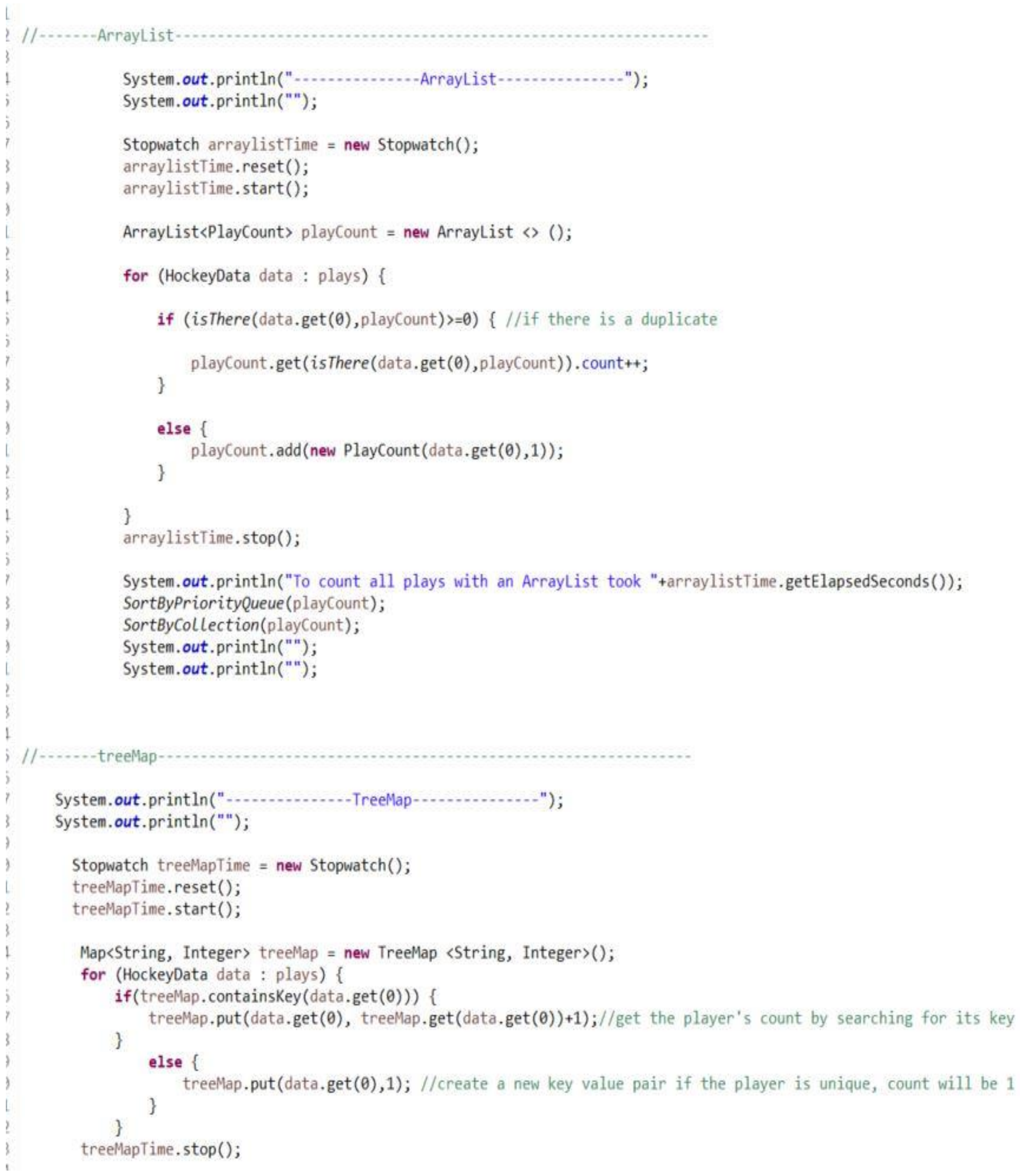
```
public static void SortByCollection (ArrayList<PlayCount> playCount) {
	System.out.println("Found " + playCount.size() + " unique players.");
	System.out.println("The 20 players with the most frequent plays and their 
counts are: ");
```

```
        Collections.sort(playCount);
        for(int i = playCount.size()-1; i>playCount.size()-21; i--) {
              System.out.println(playCount.get(i).name + "     " +
playCount.get(i).count );
        }
          }
```

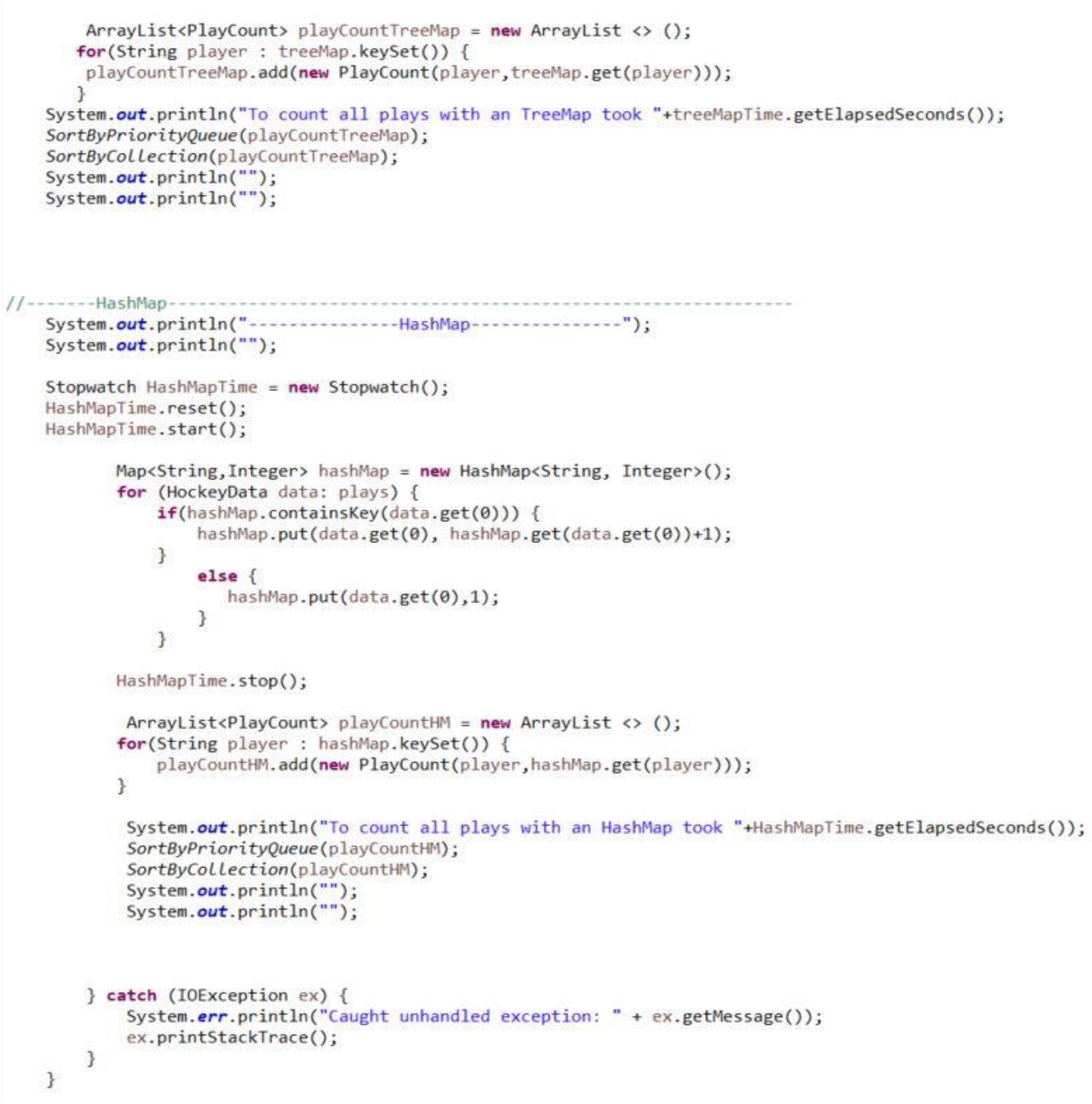
b) However, there appears to be a even more efficient to get the top 20, which is priority queue. The insertion cost of the priority queue is $O(n)$. When objects are added/constructor into the priority, they will automatically be arranged in their natural order. A new `compare` method was added. It is in fact more efficient than the previous sorting method since it performs fewer comparisons (very likely that it will do fewer than 20 comparisons for each pair). Please see below for the implementation of the priority queue:

```
public static void SortByPriorityQueue (ArrayList <PlayCount>PlayCount) {
        System.out.println("Found " + PlayCount.size() + " unique players.");
        System.out.println("The 20 players with the most frequent plays and
their counts are: ");
        PriorityQueue<PlayCount> priorityQueue = new
PriorityQueue<PlayCount>(20, new Comparator<PlayCount>() {
        public int compare(PlayCount player1, PlayCount player2) {
                if(player1.count > player2.count) {
                        return -1;
                }
                        else if(player1.count < player2.count) {
                                return 1;
                        }
                return 0;
                        }
      });
        for(PlayCount player : PlayCount) {
                priorityQueue.add(player);
        }
        for(int j=0; j<20; j++) {
                System.out.format("%d. %s    %d %n", j
1 ,priorityQueue.peek().name, priorityQueue.remove().count);
                }
     }
```

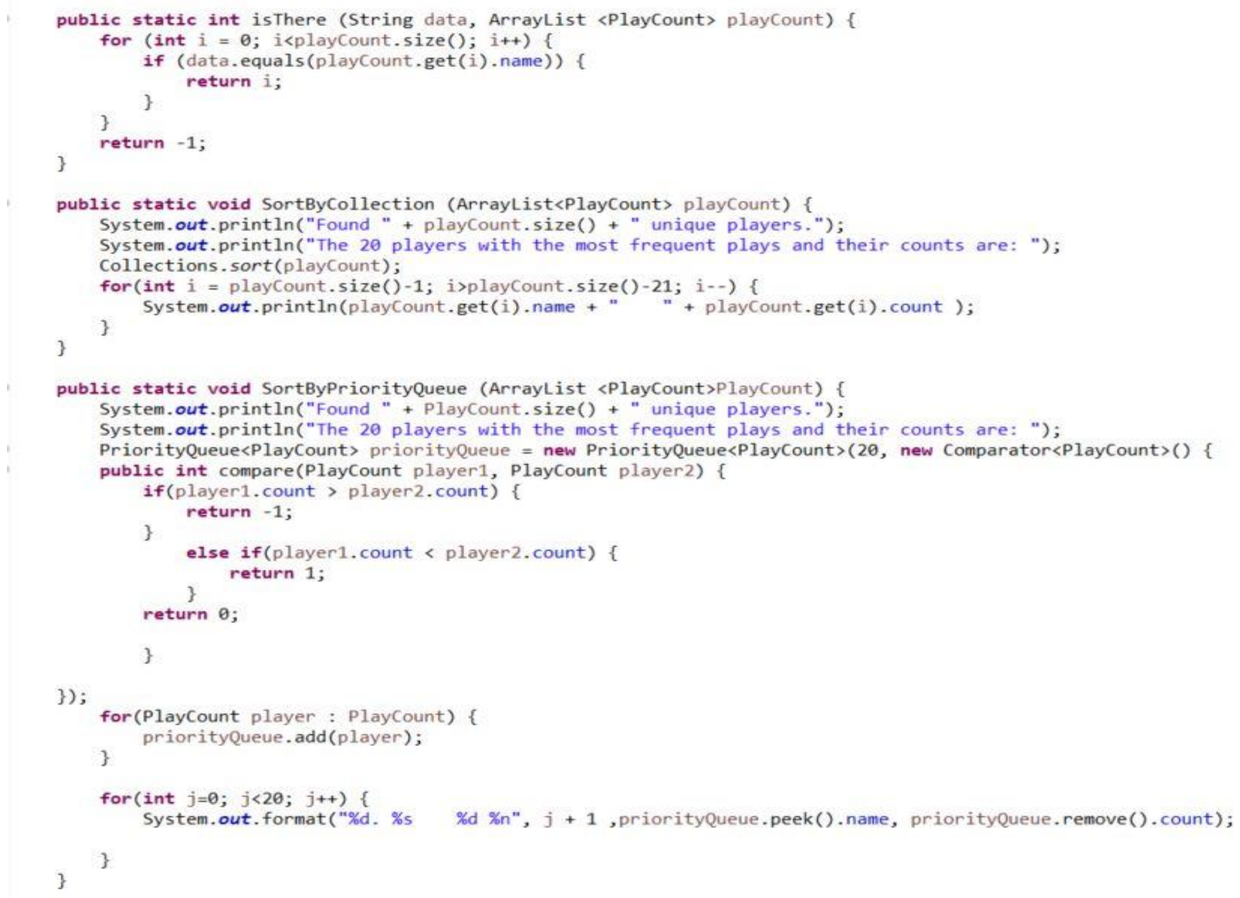
c) No issues were encountered in completing this assignment.



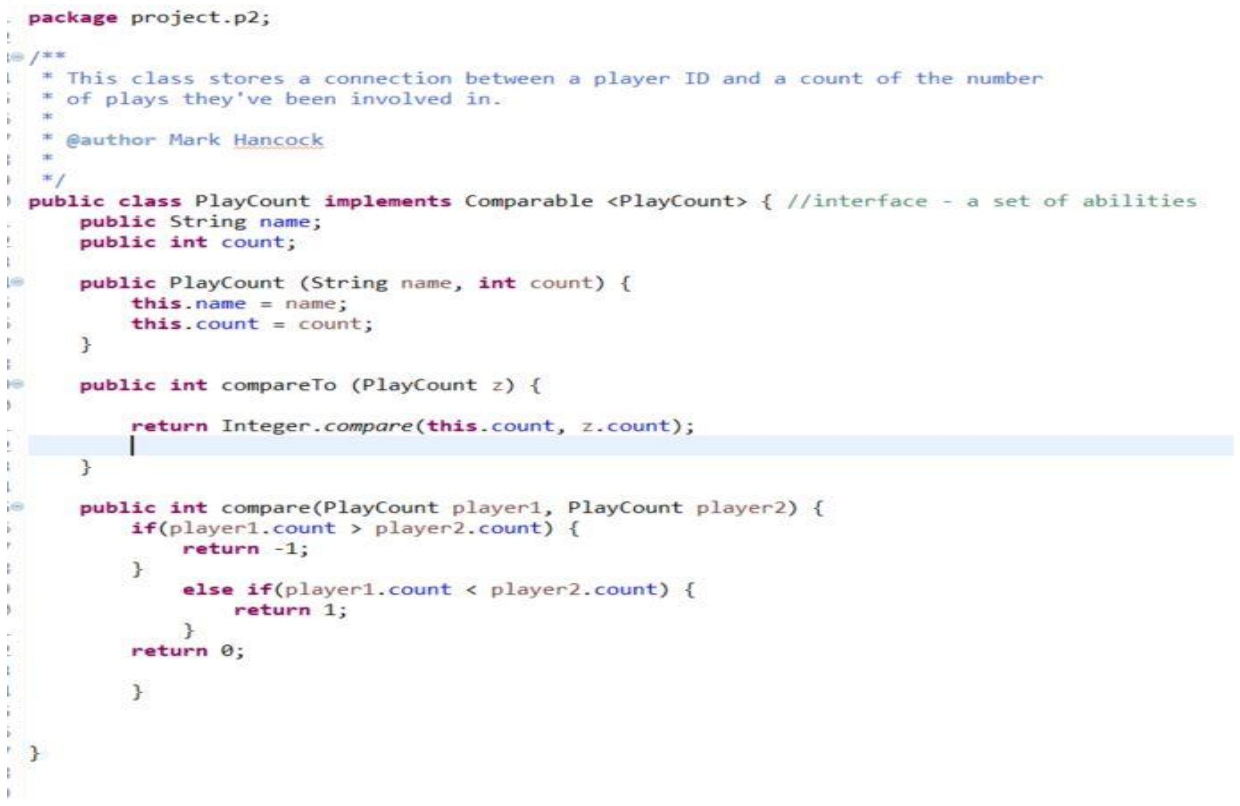
```java
//-------ArrayList---------------------------------------------------------------

            System.out.println("---------------ArrayList---------------");
            System.out.println("");

            Stopwatch arraylistTime = new Stopwatch();
            arraylistTime.reset();
            arraylistTime.start();

            ArrayList<PlayCount> playCount = new ArrayList <> ();

            for (HockeyData data : plays) {

                if (isThere(data.get(0),playCount)>=0) { //if there is a duplicate

                    playCount.get(isThere(data.get(0),playCount)).count++;
                }

                else {
                    playCount.add(new PlayCount(data.get(0),1));
                }

            }
            arraylistTime.stop();

            System.out.println("To count all plays with an ArrayList took "+arraylistTime.getElapsedSeconds());
            SortByPriorityQueue(playCount);
            SortByCollection(playCount);
            System.out.println("");
            System.out.println("");



//-------treeMap---------------------------------------------------------------

        System.out.println("---------------TreeMap---------------");
        System.out.println("");

          Stopwatch treeMapTime = new Stopwatch();
          treeMapTime.reset();
          treeMapTime.start();

           Map<String, Integer> treeMap = new TreeMap <String, Integer>();
           for (HockeyData data : plays) {
               if(treeMap.containsKey(data.get(0))) {
                   treeMap.put(data.get(0), treeMap.get(data.get(0))+1);//get the player's count by searching for its key
               }
                   else {
                       treeMap.put(data.get(0),1); //create a new key value pair if the player is unique, count will be 1
                   }
               }
           treeMapTime.stop();
```



```java
         ArrayList<PlayCount> playCountTreeMap = new ArrayList <> ();
        for(String player : treeMap.keySet()) {
         playCountTreeMap.add(new PlayCount(player,treeMap.get(player)));
        }
    System.out.println("To count all plays with an TreeMap took "+treeMapTime.getElapsedSeconds());
    SortByPriorityQueue(playCountTreeMap);
    SortByCollection(playCountTreeMap);
    System.out.println("");
    System.out.println("");



//-------HashMap-------------------------------------------------------------
    System.out.println("---------------HashMap---------------");
    System.out.println("");

    Stopwatch HashMapTime = new Stopwatch();
    HashMapTime.reset();
    HashMapTime.start();

            Map<String,Integer> hashMap = new HashMap<String, Integer>();
            for (HockeyData data: plays) {
                if(hashMap.containsKey(data.get(0))) {
                    hashMap.put(data.get(0), hashMap.get(data.get(0))+1);
                }
                    else {
                       hashMap.put(data.get(0),1);
                    }
                }

            HashMapTime.stop();

             ArrayList<PlayCount> playCountHM = new ArrayList <> ();
            for(String player : hashMap.keySet()) {
                playCountHM.add(new PlayCount(player,hashMap.get(player)));
            }

             System.out.println("To count all plays with an HashMap took "+HashMapTime.getElapsedSeconds());
             SortByPriorityQueue(playCountHM);
             SortByCollection(playCountHM);
             System.out.println("");
             System.out.println("");



        } catch (IOException ex) {
             System.err.println("Caught unhandled exception: " + ex.getMessage());
             ex.printStackTrace();
        }
    }
```



```java
public static int isThere (String data, ArrayList <PlayCount> playCount) {
    for (int i = 0; i<playCount.size(); i++) {
        if (data.equals(playCount.get(i).name)) {
            return i;
        }
    }
    return -1;
}

public static void SortByCollection (ArrayList<PlayCount> playCount) {
    System.out.println("Found " + playCount.size() + " unique players.");
    System.out.println("The 20 players with the most frequent plays and their counts are: ");
    Collections.sort(playCount);
    for(int i = playCount.size()-1; i>playCount.size()-21; i--) {
        System.out.println(playCount.get(i).name + "     " + playCount.get(i).count );
    }
}

public static void SortByPriorityQueue (ArrayList <PlayCount>PlayCount) {
    System.out.println("Found " + PlayCount.size() + " unique players.");
    System.out.println("The 20 players with the most frequent plays and their counts are: ");
    PriorityQueue<PlayCount> priorityQueue = new PriorityQueue<PlayCount>(20, new Comparator<PlayCount>() {
    public int compare(PlayCount player1, PlayCount player2) {
        if(player1.count > player2.count) {
            return -1;
        }
            else if(player1.count < player2.count) {
                return 1;
            }
        return 0;

        }

});
    for(PlayCount player : PlayCount) {
        priorityQueue.add(player);
    }

    for(int j=0; j<20; j++) {
        System.out.format("%d. %s     %d %n", j + 1 ,priorityQueue.peek().name, priorityQueue.remove().count);

    }
}
```



```java
package project.p2;

/**
 * This class stores a connection between a player ID and a count of the number
 * of plays they've been involved in.
 *
 * @author Mark Hancock
 *
 */
public class PlayCount implements Comparable <PlayCount> { //interface - a set of abilities
    public String name;
    public int count;

    public PlayCount (String name, int count) {
        this.name = name;
        this.count = count;
    }

    public int compareTo (PlayCount z) {

        return Integer.compare(this.count, z.count);

    }

    public int compare(PlayCount player1, PlayCount player2) {
        if(player1.count > player2.count) {
            return -1;
        }
            else if(player1.count < player2.count) {
                return 1;
            }
        return 0;

        }


}
```

Acknowledgment of Receiving Assistance or Use of Others' Ideas

I received the following help, assistance, or any ideas from classmates, other knowledgeable people, books or non-course websites (please include a description of discussions with the TA or the instructor):

The instruction manual written by Dr. Mark Hancock as well as the documentation links below:

1. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
2. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
3. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
4. https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

Record of Giving Assistance to Others

I gave the following help, assistance, or ideas to the following classmates (please describe what assistance to whom was given by you): N/A

Declaration

I declare that except for the assistance noted above, assistance provided on the course website, and material provided by the instructor and/or TAs that this is my original work.

I have neither given nor received an electronic or printed version of any part of this code to/from anyone.

I declare that any program output submitted as part of the assignment was generated by the program code submitted and not altered in any way.

[illegible]