Investigating Gender Imbalance in 50 Hollywood Movies

Caroline Jung, Emily Lu, and Rachel Suarez

I. Introduction

This project used real world data to investigate the gender imbalance in a set of 50 Hollywood movies. The film industry is a largely male-dominant field. Using a series of tests designed by women in the film industry, our code parsed through the data to find which movies have sufficient female representation on and off screen. The thirteen tests analyze different components of a movie to determine whether a movie passes or not, including behind the scenes crew, intersectionality, and the cast. We defined a feminist score as a movie that passes the Bechdel Test, Rees-Davies Test, Ko Test, Pierce Test, and Keoze-Dottle Test. These tests combined define what we consider the "bare-minimum" for a feminist movie. Each test has a different weight in our scoring system. If a movie passes all five of these tests, as well as any additional tests, we will consider those as well. The highest score a movie can have is 108, with 100 meaning it passed all five of the tests we selected.

The Pierce Test identifies movies with a female character who is a protagonist or antagonist with her own story, and has dimension and exists authentically with needs and desires that she pursues through dramatic action. The audience must be able to empathize with her actions and desires. We gave this test a weight of 28, as there are many qualifications to be able to pass this test. The Ko Test identifies movies with a non-white passing female character who speaks in five or more scenes. We gave this test a score of 25, as casting non-white women is very important for representation on and off screen. The Koeze-Dottle Test identifies movies with at least 50% women in the supporting cast. We gave this test a weight of 21, as women should be represented equally. The Rees-Davies Test identifies movies with two women in each department. We gave this test a weight of 14, as having only two women in each department is a minimal requirement. The Bechdel Test is defined as a movie with two named female characters that have a conversation that is not about a man. We gave this test a weight of 12 points as this is a very minimal requirement.

II. Method

To calculate our feminist score, we created a method that iterates through the vector that holds a movie's scores. First, we check the indexes that hold the tests we've selected as crucial in calculating the feminist score: the Bechdel (index 0), Pierce (index 1), Ko (index 6), Koeze-Dottle (index 9) and Rees-Davies (index 12) tests. Iterating through the vector of test results, if the test result for each specified index (mentioned above) holds a 0, meaning that the movie passed that specific test, we add points to our *count* object. Each test has a specified number of points or "weight" depending on how important we have deemed it. For example, we look at index 0 to see if the movie has passed the Bechdel test. If there is a 0 in that index, that indicates that the movie has passed the Bechdel test, so we add 12, the specified weight for that test, to our *count* object. We repeat this process to find if the movie has passed our other selected

tests. Out of a 100 point scale, the Bechdel test is worth 12 points, Rees-Davies is 14, Koeze-Dottle is 21, Ko is 25, and Pierce is 28. It is important to note however, that the maximum number of points a movie may have is actually 108. This is possible since after checking each index for the tests we selected, we iterate through the vector to check for any additional tests a movie may have passed. To make sure we don't add a score for the tests we already checked for, there is an *additional* boolean that is only true if the index is not 0, 1, 6, 9, or 12, the index values we have already checked in the vector. If *additional* is true and the vector value is 0, we add 1 to our *count*. This method then returns *count*, the final feminist score for the movie, once it has iterated through the entire test results vector.

We gave each test a different weight so that there would be less ties to break when sorting by highest score. Our solution would only change if there were more tests being performed for each movie. If there are additional movies added to our data set, our feministScore method will still work.

Our compareTo() method compares the feminist scores of two movies to decide which of the two is smaller. We decided to break ties between movies with the same feminist score by comparing the Hashcodes of their movie titles such that this compareTo() method never returns "0." We implemented our PriorityQueue using a LinkedMaxHeap, which uses heapifyAdd() and heapifyRemove(). These methods use compareTo() to reconfigure the heap after an element is added or removed to make sure that the MaxHeap is oriented correctly. That is, each parent node is greater than both its children. Thus, writing a compareTo() method that never evaluates two Movie objects as equal breaks potential ties in the PriorityQueue.

III. Results

The results of our analysis is below, ranking the movies with the most female representation to movies with the least amount of female representation.

Sing: 69 actors
Bad Moms: 16 actors
Independence Day: Resurgence: 28 actors
Boo! A Madea Halloween: 42 actors
Sausage Party: 33 actors
Suicide Squad: 54 actors
Ice Age: Collision Course: 18 actors
Hidden Figures: 96 actors
Ghostbusters: 54 actors
Allegiant: 91 actors
Star Trek Beyond: 23 actors
Fantastic Beasts and Where to Find Them: 61 actors
The Purge: Election Year: 41 actors
Moana: 7 actors
Storks: 25 actors
The Girl on the Train: 20 actors
The Boss: 37 actors

Arrival: 60 actors
La La Land: 49 actors
The Conjuring 2: The Enfield Poltergeist: 32 actors
Don't Breathe: 10 actors
Lights Out: 14 actors
Trolls: 32 actors
Finding Dory: 28 actors
Alice Through the Looking Glass: 23 actors
Now You See Me 2: 48 actors
Captain America: Civil War: 26 actors
Ride Along 2: 50 actors
Central Intelligence: 25 actors
Kung Fu Panda 3: 36 actors
The Accountant: 58 actors
Passengers: 27 actors
Miss Peregrine's Home for Peculiar Children: 53 actors
Batman v Superman: Dawn of Justice: 122 actors
X-Men: Apocalypse: 90 actors
Jason Bourne: 43 actors
Zootopia: 32 actors
10 Cloverfield Lane: 10 actors
Rogue One: A Star Wars Story: 33 actors
The Magnificent Seven: 48 actors
The Legend of Tarzan: 58 actors
The Secret Life of Pets: 31 actors
Pete's Dragon: 41 actors
The Angry Birds Movie: 47 actors
The Jungle Book: 11 actors
Sully: 61 actors
Deadpool: 34 actors
Hacksaw Ridge: 90 actors
Teenage Mutant Ninja Turtles: Out of the Shadows: 23 actors
Doctor Strange: 29 actors

IV. Conclusion

Looking at the results, we were initially surprised that *Hidden Figures* is not as high up on the feminist score scale because it is a classic when it comes to representing Black women in STEM. However, we hypothesized the reason why it has a lower feminist score than initially thought may be due to the fact that the movie was set in the 1940s, a time when white men dominated the STEM industry as well as social stigmas about higher education for women.

We were also surprised that *Sausage Party* is the fifth most feminist movie in the list considering its raunchy plotline and the fact that the movie has received backlash for its racism and sexism. However, there has also been debates about the racism and sexism prevalent in the film since these could be viewed as a critique of society when viewed as satirical.

We were not surprised that the Marvel movie, *Dr. Strange*, is ranked the lowest on the list, as Marvel movies are infamous for having poor female representation.

As a new additional test, we think that we also need to evaluate the movie's director, more specifically, that the director be a woman. We think this new test is valuable because with the increased power that a woman would have when directing a movie, they would also be able to bring nuances of women's experiences and struggles to light even when it is not the main focus of the movie. Also, to create a movie is to present the world through multiple lenses, so to have a female director is to present a worldview that represents a woman's experiences.

V. Code

We summarize the code that was written for this project below, sub sectioned by each significant class.

Actor Class:

```
/**
 * Represents an object of type Actor. An Actor has a name and a gender.
 *
 * @author (Provided code Stella K., Caroline Jung)
 * @version (May 12, 2022)
 */
public class Actor
{
   private String name;
   private String gender;

   /**
    * Constructor for class Actor.
    */
   public Actor(String name, String gender){
      this.name = name;
      this.gender = gender;
   }

   /**
    * This method is defined here because Actor (mutable) is used as a key in a Hashtable.
    * It makes sure that same Actors have always the same hash code.
    * So, the hash code of any object that is used as key in a hash table,
    * has to be produced on an *immutable* quantity,
    * like a String (such a string is the name of the actor in our case)
    *
    * @return an integer, which is the hash code for the name of the actor
    */
   public int hashCode() {
      return name.hashCode();
   }
```

```
/**
 * Tests this actor against the input one and determines whether they are equal.
 * Two actors are considered equal if they have the same name and gender.
 *
 * @return true if both objects are of type Actor,
 * and have the same name and gender, false in any other case.
 */
public boolean equals(Object other) {
   if (other instanceof Actor) {
      return this.name.equals(((Actor) other).name) &&
      this.gender.equals(((Actor) other).gender); // Need explicit (Actor) cast to use .name
   } else {
      return false;
   }
}

/**
 * Getter method for gender instance variable.
 *
 * @return The gender of this actor.
 */
public String getGender(){
   return gender;
}

/**
 * Getter method for name instance variable.
 *
 * @result The name of this actor.
 */
public String getName(){
   return name;
}

/**
 * Setter method for gender instance variable.
 *
 * @param g The gender of this actor.
 */
public void setGender(String g){
   gender = g;
}

/**
 * Setter method for name instance variable.
```

```
     *
     * @param n The name of this actor.
     */
    public void setName(String n){
        name = n;
    }

    /**
     * Returns a String representation of object Actor.
     *
     * @return A String representation of this actor, containing the actor's name and gender.
     */
    public String toString(){
        return name + ": " + gender;
    }

    /**
     * Main method for testing.
     */
    public static void main(String[] args){
        System.out.println("Testing constructor, toString(), hashCode():");
        Actor a1 = new Actor("Scarlett Johansson", "Female");
        System.out.println("Actor 1: " + a1);
        System.out.println("Hashcode: " + a1.hashCode());
        System.out.println();

        System.out.println("Testing equals():");
        Actor a2 = new Actor("Scarlett Johansson", "Female"); //same as a1
        Actor a3 = new Actor("Ryan Gosling", "Male");
        System.out.println("Actor 1: " + a1);
        System.out.println("Actor 2: " + a2);
        System.out.println("Actor 3: " + a3);
        System.out.println("Comparing actor 1 & 2. EXPECTED: true | ACTUAL: " + a1.equals(a2));
        System.out.println("Comparing actor 2 & 3. EXPECTED: false | ACTUAL: " + a2.equals(a3));
    }
}
```

Movie Class:

```
import java.util.Hashtable;
import java.util.Vector;
import java.util.Scanner;
import java.io.IOException;
import java.io.File;
import java.util.LinkedList;
```

```
/**
 * Represents an object of type Movie.
 * A Movie object has a title, some Actors, and results for the twelve Bechdel tests.
 *
 * @author (Provided code Stella K., Caroline Jung, Emily Lu, Rachel Suarez)
 * @version (May 12, 2022)
 */
public class Movie implements Comparable<Movie>
{
    private String title;
    private Hashtable<Actor,String> roles;
    private Vector<String> testResults;

    /**
     * Constructor for class Movie.
     */
    public Movie(String title){
        this.title = title;
        //initialize other to empty structures
        roles = new Hashtable<Actor,String>();
        testResults = new Vector<String>();
    }

    /**
     * Compares two movie objects' feminist scores.
     * @param movie the movie to compared to
     * @return a negative integer if this movie's feminist score is less than the specified movie's
     * score and a positive integer if this movie's feminist score is more than the specified movie's
     * score. If this movie's feminist score is equal to the specified movie's score, we compare the
     * movies' titles' hashcode to break the tie (smaller hashcode is deemed "smaller than" the other
     * specified movie).
     */
    public int compareTo(Movie movie){
        int result = this.feministScore() - movie.feministScore();
        if (result == 0){
            //hashcodes for titles of movies are all diff
            if (this.title.hashCode() < movie.title.hashCode())
                result = -1;
            else
                result = 1;
        }
        return result;
        //negative if this < movie
        //positive if this > movie
        //breaks ties by comparing hashcode of the movie's titles
```

```
}

/**
 * Getter method for title instance variable.
 * @return The title of this movie.
 */
public String getTitle(){
   return title;
}

/**
 * Getter method for roles instance variable of a hashtable of an actor and their role.
 * @return A Hashtable with all actors who played in this movie.
 */
public Hashtable<Actor,String> getAllActors(){
   return roles;
}

/**
 * Returns a LinkedList with names of all actors who played in this movie.
 * @return A LinkedList with names of all actors who played in this movie.
 */
public LinkedList<String> getActors(){
   LinkedList<String> allActors = new LinkedList<String>();
   // add each key:value pair in this movie's Hashtable of Actors to a LinkedList
   roles.forEach((key, value) -> {
         allActors.add(key.getName());
      });
   return allActors;
}

/**
 * Getter method for result instance variable of vector of all the results.
 * @return A Vector with the Bechdel test results for this movie,
 * where "1" and "0" are used to indicate whether this movie passed the corresponding test.
 */
public Vector<String> getAllTestResult(){
   return testResults;
}

/**
 * Takes a String of 1's and 0's and initializes the Vector of test results for this movie.
 * @param results A String of 1's and 0's, each representing the result of a Bechdel-
 * alternative test on this movie.
 */
public void setTestResults(String results){
```

```
    String[] splitResults = results.split(",");
    for(int i=0; i<splitResults.length; i++){
        testResults.add(splitResults[i]);
    }
}

/**
 * Tests this movie object with the input one and determines whether they are equal.
 * @return true if both objects are movies and have the same title,
 * false in any other case.
 */
public boolean equals(Object other) {
    if (other instanceof Movie) {
        return this.title.equals(((Movie) other).title); // Need explicit (Movie) cast to use .title
    } else {
        return false;
    }
}

/**
 * Takes a String of data on a role an actor played in the format of
 * '"MOVIE","ACTOR","CHARACTER_NAME","TYPE","BILLING","GENDER"' and
 * adds an entry to the Hashtable of Actors for this movie.
 * @param line A String of data on a role an actor played.
 * @return An object representing the actor that the input String pertains to.
 */
public Actor addOneActor(String line){
    String[] splitLine = line.split(",");
    // create an Actor object to be key
    Actor actor = new Actor(splitLine[1].substring(1, splitLine[1].length() - 1),
            splitLine[5].substring(1, splitLine[5].length() - 1));
    // get type of role as value
    String role = splitLine[3].substring(1, splitLine[3].length() - 1);
    // add actor,
    roles.put(actor, role);
    return actor;
}

/**
 * Takes the name of an input file and
 * adds all its Actors to this movie's Hashtable of Actors.
 * @param actorsFile Input file containing data on actors and their roles.
 */
public void addAllActors(String actorsFile){
    try{
        Scanner fileScan = new Scanner(new File(actorsFile));
```

```
        // skip header line
        fileScan.nextLine();
        while(fileScan.hasNextLine()){
            String tempLine = fileScan.nextLine();
            if(tempLine.contains(title)){
                addOneActor(tempLine);
            }
        }
        fileScan.close();
    }
    catch(IOException ex){
        System.out.println(ex);
    }
}

/**
 * Returns a toString() representation of this movie,
 * containing the title of the movie and the number of actors.
 * @return String representation of this movie.
 */
public String toString(){
    return title + ": " + roles.size() + " actors";
}

/**
 * Calculates the feminist score for a movie.
 *
 * @return int the number representing the feminist score
 */
public int feministScore(){
    //count: add more points if they passed depending on weight of the test, 100 point scale
    int count = 0;
    //go through vector: if 0, then add score. if 1, do nothing
    //bechdel: i=0        weight: 12
    //rees-davies: i=12   weight: 14
    //ko: i=6             weight: 25
    //pierce: i=1         weight: 28
    //koeze-dottle: i=9   weight: 21

    if (testResults.get(0).equals("0"))//bechdel
        count += 12;
    if (testResults.get(12).equals("0"))//rees-davies
        count += 14;
    if (testResults.get(6).equals("0"))//ko
        count += 25;
    if (testResults.get(1).equals("0"))//pierce
```

```
        count += 28;
    if (testResults.get(9).equals("0"))//koeze-dottle
        count += 21;

    for (int i=0; i<testResults.size(); i++){
        //for each additional test that it passes, add 1 point
        //max score possible: 108
        boolean additional = (i==2) ||(i==3) ||(i==4) ||(i==5) ||(i==7) ||(i==8) ||(i==10) ||(i==11);
        if (additional && testResults.get(i).equals("0"))
            count += 1;
    }
    return count;
}

/**
 * Setter method for title instance variable.
 * @param t the string to set the title to
 */
public void setTitle(String t){
    title = t;
}

/**
 * Setter method for roles instance variable of a hashtable of an actor and their role.
 * @param r a hashtable of actors and their role to set the hashtable to
 */
public void setRole(Hashtable<Actor,String> r){
    roles = r;
}

/**
 * Setter method for result instance variable of vector of all the results.
 * @param res a vector of strings to set the vector of all results to
 */
public void setResult(Vector<String> res){
    testResults = res;
}

/**
 * Main method for testing.
 */
public static void main(String[] args){
    System.out.println("Testing addOneActor() and addAllActors()");
    Movie alpha = new Movie("Alpha");
    alpha.addAllActors("data/small_castGender.txt");
    System.out.println("\nPrinting roles in Alpha:"
```

```
        + "\nEXPECTED: " +
        "{Patrice Lovely | Female=Supporting, Cassi Davis | Female=Supporting, Stella |
Male=Leading, Tyler Perry | Male=Leading}"
        + "\nACTUAL: " + alpha.roles);
    Movie beta = new Movie("Beta");
    beta.addAllActors("data/small_castGender.txt");
    System.out.println("\nPrinting roles in Beta:"
        + "\nEXPECTED: " +
        "{Takis | Female=Supporting, Patrice Lovely | Female=Supporting, Cassi Davis |
Female=Supporting, Tyler Perry | Male=Leading}"
        + "\nACTUAL: " + beta.roles);
    Movie gamma = new Movie("Gamma");
    gamma.addAllActors("data/small_castGender.txt");
    System.out.println("\nPrinting roles in Gamma:"
        + "\nEXPECTED: " +
        "{Cassi Davis | Female=Supporting, Tyler Perry | Male=Leading}"
        + "\nACTUAL: " + gamma.roles);
    Movie lambda = new Movie("Lambda");
    lambda.addAllActors("data/small_castGender.txt");
    System.out.println("\nPrinting roles in Lambda:"
        + "\nEXPECTED: {}" +
        "\nACTUAL: " + lambda.roles);
    System.out.println("\nReading from \"nonexistent.txt\"");
    System.out.print("EXPECTED: FileNotFoundException" + "\nACTUAL: ");
    lambda.addAllActors("data/nonexistent.txt");
    System.out.println("\nTesting toString()");
    System.out.println("\nCalling toString() on Alpha:" + "\nEXPECTED: " + "Alpha | 4 actors" +
"\nACTUAL: " + alpha);
    System.out.println("\nCalling toString() on Beta:" + "\nEXPECTED: " + "Beta | 4 actors" +
"\nACTUAL: " + beta);
    System.out.println("\nCalling toString() on Gamma:" + "\nEXPECTED: " + "Gamma | 2 actors"
+ "\nACTUAL: " + gamma);
    System.out.println("\nTesting getActors()");
    System.out.println("\nCalling getActors() on Alpha:" + "\nEXPECTED: " + "[Tyler Perry,
Stella, Cassi Davis, Patrice Lovely]" + "\nACTUAL: " + alpha.getActors());
    System.out.println("\nCalling getActors() on Gamma:" + "\nEXPECTED: " + "[Tyler Perry,
Cassi Davis]" + "\nACTUAL: " + gamma.getActors());
    System.out.println("\nTesting setTestResults()");
    alpha.setTestResults("0,0,0,1,0,0,0,1,0,0,1,1,1");
    System.out.println("Printing test results for Alpha:" + "\nEXPECTED: [0, 0, 0, 1, 0, 0, 0, 1, 0,
0, 1, 1, 1]" + "\nACTUAL: " + alpha.testResults);
    beta.setTestResults("1,1,0,1,0,1,0,1,1,0,1,0,0");
    System.out.println("Printing test results for Beta:" + "\nEXPECTED: [1, 1, 0, 1, 0, 1, 0, 1, 1, 0,
1, 0, 0]" + "\nACTUAL: " + beta.testResults);
    System.out.println("\nTesting feministScore()");
    System.out.println("Testing Alpha: EXPECTED: 90" + " ACTUAL: " + alpha.feministScore());
```

```
        Movie fem = new Movie("fem");
        System.out.println("Test results for fem: [0,0,0,0,0,0,0,0,0,0,0,0,0]");
        fem.setTestResults("0,0,0,0,0,0,0,0,0,0,0,0,0");
        System.out.println("Testing fem: EXPECTED: 108" + " ACTUAL: " + fem.feministScore());
        System.out.println("\nTesting compareTo()");
        System.out.println("Testing alpha and fem scores: EXPECTED: -18 | ACTUAL: " +
alpha.compareTo(fem));
    }
}
```

MovieCollection class:

```
import java.util.LinkedList;
import java.io.IOException;
import java.io.File;
import java.util.Scanner;
import java.util.Hashtable;
import java.util.Vector;
import java.io.PrintWriter;
import javafoundations.PriorityQueue;

/**
 * Represents an object of type MovieCollection.
 * Manages collections of all movies and all actors given a file with movies with their
 * test results and actors' information.
 * @author (Caroline Jung, Emily Lu, Rachel Suarez)
 * @version (May 12, 2022)
 */
public class MovieCollection
{
    // instance variables
    private LinkedList<Movie> allMovies;
    private LinkedList<Actor> allActors;

    /**
     * Constructor for class MovieCollection.
     */
    public MovieCollection(String testsFileName, String castsFileName){
        allMovies = new LinkedList<Movie>();
        this.readMovies(testsFileName);
        allActors = new LinkedList<Actor>();
        this.readCasts(castsFileName);
    }

    /**
```

```
 * Reads the input file, and uses its first column (movie title) to create all movie objects.
 * Adds the included information on the Bachdel test results to each movie.
 * @param fName the name of the file to be read
 */
private void readMovies(String fName){
   try{
      Scanner fileScan = new Scanner(new File(fName));
      fileScan.useDelimiter(",");
      // skip header line
      fileScan.nextLine();

      while(fileScan.hasNextLine()){
         String name = fileScan.next();

         String testResult = fileScan.nextLine().substring(1);

         // create new movie object and populate test results
         Movie m = new Movie(name);
         m.setTestResults(testResult);

         allMovies.add(m);
      }
      fileScan.close();
   }
   catch(IOException ex){
      System.out.println(ex);
   }
}

/**
 * Reads the casts for each movie, from input casts file; Assume lines in this file are formatted.
 * If movie doesn't have test results, it is not included in the collection. There are no repeats in
 * Movies.
 * @param fName the name of the file to be read.
 */
private void readCasts(String fName){
   for (int i=0; i<allMovies.size(); i++){
      if(!allMovies.get(i).getAllTestResult().isEmpty()){ // ignore movies with no test results
         allMovies.get(i).addAllActors(fName); //populates hashtable of actors for a movie
         Hashtable<Actor,String> hash = allMovies.get(i).getAllActors();
         hash.forEach((key, value) ->
            {
               if (!allActors.contains(key)){ //check if Actor is already in the linked list
                  allActors.add(key);
               }
            });
```

```
        }
    }
}

/**
 * Getter method for the allActors Linked List.
 * @return LinkedList<Actor> linked list of all actors
 */
public LinkedList<Actor> getActors(){
    return allActors;
}

/**
 * Getter method for the allMovies Linked List.
 * @return LinkedList<Actor> linked list of all movies
 */
public LinkedList<Movie> getMovies(){
    return allMovies;
}

/**
 * Getter method for the accessing all movie titles.
 * @return LinkedList<String> linked list of all movie titles
 */
public LinkedList<String> getMovieTitles(){
    LinkedList<String> movieTitles = new LinkedList<String>();
    for (int i = 0; i< allMovies.size(); i++){
        movieTitles.add(allMovies.get(i).getTitle());
    }
    return movieTitles;
}

/**
 * Getter method for the accessing all actor names.
 *
 * @return LinkedList<String> linked list of all actor names
 */
public LinkedList<String> getActorNames(){
    LinkedList<String> actorNames = new LinkedList<String>();
    for (int i = 0; i<allActors.size(); i++){
        actorNames.add(allActors.get(i).getName());
    }
    return actorNames;
}

/**
```

```
    * Returns a list of all Movies that pass the n-th Bechdel test
    * @param n an integer indicating which nth Bechdel test to examine
    * @return LinkedList<Movie> linked list of movies that passed the nth Bechdel test
    */
   public LinkedList<Movie> findAllMoviesPassedTestNum(int n){
      LinkedList<Movie> passed = new LinkedList<Movie>();
      for (int i = 0; i < allMovies.size(); i++){
         if (allMovies.get(i).getAllTestResult().get(n).equals("0")){
            //each movie's test result for n is considered to pass (0)
            passed.add(allMovies.get(i));
         }
      }
      return passed;
   }

   /**
    * Writes all the movies that passed the following criteria to a separate file: 1) Bechdel test
    * 2) Peirce or Landau test, 3) passed White test but failed Rees-Davies test.
    * File contains the number of movies from each result and the movie titles.
    * @param outFileName the name of the file to write the results to
    */
   private void writeAllMoviesPassedCriteria(String outFileName){
      try{
         PrintWriter writer = new PrintWriter(new File(outFileName));
         /*
          * write all movies that passed Bechdel test to file
          */
         LinkedList<Movie> passedBechdel = findAllMoviesPassedTestNum(0);
         writer.println("Movies that passed the Bechdel test (" + passedBechdel.size() + " total):");
         for (int i = 0; i < passedBechdel.size(); i++){
            writer.println(passedBechdel.get(i).getTitle()); // write title of current movie to file
         }

         /*
          * write all movies that passed Peirce or Landau tests to file
          */
         LinkedList<Movie> passedPeirceOrLandau = findAllMoviesPassedPeirceOrLandau();
         writer.println("\nMovies that passed the Peirce or Landau test (" +
passedPeirceOrLandau.size() + " total):");
         for (int i = 0; i< passedPeirceOrLandau.size(); i++){
            writer.println(passedPeirceOrLandau.get(i).getTitle()); // write title of current movie to file
         }

         /*
          * write all movies that passed the White test but failed the Rees-Davies test to file
          */
```

```
        LinkedList<Movie> passedWhiteFailedReesDavies =
findAllMoviesPassedWhiteFailedReesDavies();
        writer.println("\nMovies that passed the White test but failed the Rees-Davies test (" +
passedWhiteFailedReesDavies.size() + " total):");
        for (int i = 0; i< passedWhiteFailedReesDavies.size(); i++){
           writer.println(passedWhiteFailedReesDavies.get(i).getTitle()); // write title of current movie
to file
        }
        writer.close();
     }
     catch(IOException ex){
        System.out.println(ex);
     }
  }

  /**
   * Returns all the movies (in a linked list) that passed the Peirce or Landau test.
   * @return  LinkedList<Movie> the movies that passed the Peirce or Landau test
   */
  private LinkedList<Movie> findAllMoviesPassedPeirceOrLandau(){
     LinkedList<Movie> passed = new LinkedList<Movie>();
     for (int i = 0; i < allMovies.size(); i++){
        Vector<String> aTestResult = allMovies.get(i).getAllTestResult(); // get allTestResult for
current movie
        if (aTestResult.get(1).equals("0") || aTestResult.get(2).equals("0")){ // if movie passed Peirce
or Landau,
           passed.add(allMovies.get(i)); // then add it to LinkedList
        }
     }
     return passed;
  }

  /**
   * Returns all the movies (in a linked list) that passed the White test but failed the Rees-Davies.
   * @return  LinkedList<Movie> the movies that passed the White test but failed the Rees-Davies
   */
  private LinkedList<Movie> findAllMoviesPassedWhiteFailedReesDavies(){
     LinkedList<Movie> passed = new LinkedList<Movie>();
     for (int i = 0; i < allMovies.size(); i++){
        Vector<String> aTestResult = allMovies.get(i).getAllTestResult(); // get allTestResult for
current movie
        if (aTestResult.get(11).equals("0") && !aTestResult.get(12).equals("0")){ // if movie passed
White but failed Rees-Davies,
           passed.add(allMovies.get(i)); // then add it to LinkedList
        }
     }
```

```
        return passed;
    }

    /**
     * returns a PriorityQueue of movies in the provided data based on their feminist score.
     * If all movies are enqueued, they will be dequeued in priority order: from most
     * feminist to least feminist.
     * @return priority queue of movies in order of their feminist score
     */
    public PriorityQueue<Movie> prioritizeMovies(){
        PriorityQueue<Movie> pq = new PriorityQueue<Movie>();

        for (Movie movie : allMovies){
            //ties broken by the compareTo method
            pq.enqueue(movie);
        }
        return pq;
    }

    /**
     * returns a PriorityQueue of feminist scores for all movies in the provided data in order
     * of most feminist (higher score) to least feminist. Used for testing purposes.
     * @return priority queue of feminist scores in order
     */
    private PriorityQueue<Integer> testingPQ(){
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

        for (Movie movie : allMovies){
            //ties broken by the compareTo method
            pq.enqueue(movie.feministScore());
        }
        return pq;
    }

    /**
     * String representation of a MovieCollection which includes the number of movies and the movies
themselves.
     *
     * @return String string representation of a MovieCollection
     */
    public String toString(){

        String s = "This MovieCollection has " + allMovies.size() + " movies: \n";
        for (int i=0; i<allMovies.size(); i++){
            s += allMovies.get(i) + "\n";
        }
```

```
        return s;
    }

    /**
     * Main method for testing.
     */
    public static void main(String[] args){
        System.out.println("Testing readMovies() and readCast()");
        MovieCollection m1 = new MovieCollection("data/small_allTests_Copy.txt",
"data/small_castGender.txt");
        //MovieCollection m1 = new MovieCollection("data/nextBechdel_allTests.txt",
"data/nextBechdel_castGender.txt");
        System.out.println("\nPrinting all movies in small MovieCollection: \nEXPECTED: [Alpha: 4
actors, Beta: 4 actors, Gamma: 4 actors]" +
                        "\nACTUAL: " + m1.allMovies);
        System.out.println("\nPrinting all actors in small MovieCollection: \nEXPECTED: [Tyler Perry:
Male, Stella: Male, Cassi Davis: Female, Patrice Lovely: Female, Takis: Female]" +
        "\nACTUAL: " + m1.allActors);

        System.out.println("\nTesting toString(): ");
        System.out.println(m1);

        System.out.println("\nTesting findAllMoviesPassedTestNum()");
        System.out.println("\nPrinting all movies that passed test no. 1"
                        + "\nEXPECTED: [Alpha: 4 actors, Beta: 4 actors]"
                        + "\nACTUAL: " + m1.findAllMoviesPassedTestNum(0));
        System.out.println("\nPrinting all movies that passed test no. 13"
                        + "\nEXPECTED: []"
                        + "\nACTUAL: " + m1.findAllMoviesPassedTestNum(12));

        System.out.println("\nTesting findAllMoviesPassedPeirceOrLandau()");
        System.out.println("\nEXPECTED: [Beta: 4 actors, Gamma: 2 actors]" +
                        "\nACTUAL: " + m1.findAllMoviesPassedPeirceOrLandau());
        System.out.println("\nTesting findAllMoviesPassedWhiteFailedReesDavies()");
        System.out.println("\nEXPECTED: [Beta: 4 actors]" +
                        "\nACTUAL: " + m1.findAllMoviesPassedWhiteFailedReesDavies());
        m1.writeAllMoviesPassedCriteria("data/write_Test.txt");


        MovieCollection m2 = new MovieCollection("data/nextBechdel_allTests.txt",
"data/nextBechdel_castGender.txt");
        m2.writeAllMoviesPassedCriteria("data/bechdelProject_testing.txt");

        System.out.println("\nPriority queue of movies:\n" + m2.prioritizeMovies());
        System.out.println("Order of the feminist scores for all movies in the priority queue:\n" +
m2.testingPQ());
```

```
    }
}
```

PriorityQueue class:

```
package javafoundations;
import javafoundations.exceptions.*;
/**
 * Implements the Queue interface, based on a LinkedMaxHeap which implements the MaxHeap
 * interface.
 * Of type Comparable objects

 * @author (Caroline Jung, Emily Lu)
 * @version (May 12, 2022)
 */
public class PriorityQueue<T extends Comparable<T>> implements Queue<T>

{
   private LinkedMaxHeap<T> heap;

   /**
    * Constructor for priority queue class.
    */
   public PriorityQueue(){
      heap = new LinkedMaxHeap<T>();
   }

   /**
    * Adds the specified element to the rear of the queue.
    * @param element the element to be added
    */
   public void enqueue (T element){
      heap.add(element);
   }

   /**
    * Removes and returns the element at the front of the queue.
    * @return the element at the front of the queue
    */
   public T dequeue(){
      try{
         return heap.removeMax();
      }catch(EmptyCollectionException ex){
         throw new EmptyCollectionException ("Cannot dequeue from an empty priority queue.");
      }
```

```
}

/**
 * Returns a reference to the element at the front of the queue without removing it.
 * @return the reference to the element at the front of the queue
 */
public T first(){
   try{
      return heap.getMax();
   }catch(EmptyCollectionException ex){
      throw new EmptyCollectionException ("Cannot get element from an empty priority queue.");
   }
}

/**
 * Returns true if the queue contains no elements and false otherwise.
 * @return true if the queue is empty, false otherwise.
 */
public boolean isEmpty(){
   return heap.isEmpty();
}

/**
 * Returns the number of elements in the queue.
 * @return the number of elements in the queue
 */
public int size(){
   return heap.size();
}

/**
 * Returns a string representation of the queue.
 * @return string representation
 */
public String toString(){
   LinkedMaxHeap<T> temp = new LinkedMaxHeap<T>();

   String s = "<Priority Queue>: \n";
   for(T el : heap){
      T currentMax = this.dequeue();
      temp.add(currentMax);
      s += currentMax + "\n";
   }
   heap = temp;
   return s;
}
```

```
    /**
     * main method for testing.
     */
    public static void main(String[] args){
        PriorityQueue q1 = new PriorityQueue<Integer>();
        //no elements
        System.out.println(q1);
        System.out.println("Testing isEmpty(): \nEXPECTED: true | ACTUAL: " + q1.isEmpty());
        try{
            System.out.println("Testing dequeue(): \nEXPECTED: error | ACTUAL: " + q1.dequeue());
        }catch(EmptyCollectionException ex){
            System.out.println("Testing dequeue(): \n" + ex);
        }
        try{
            System.out.println("Testing first(): \nEXPECTED: error | ACTUAL: " + q1.first());
        }catch(EmptyCollectionException ex){
            System.out.println("Testing first(): \n" + ex);
        }

        System.out.println("\n-----------------------------------------------------------------");
        PriorityQueue q2 = new PriorityQueue<Integer>();
        q2.enqueue(7);
        System.out.print(q2);
        q2.enqueue(9);
        System.out.println("added 9 to queue");
        System.out.println(q2);
        System.out.println("Testing first(): \nEXPECTED: 9 | ACTUAL: " + q2.first());
        System.out.println("Testing dequeue(): \nEXPECTED: 9 | ACTUAL: " + q2.dequeue());
        System.out.println("\nQueue after dequeuing 9\n" + q2);

        System.out.println("\n-----------------------------------------------------------------");
        PriorityQueue q3 = new PriorityQueue<Integer>();
        q3.enqueue(5);
        q3.enqueue(6);
        q3.enqueue(12);
        q3.enqueue(1);
        System.out.println(q3);
        System.out.println("Testing size(): \nEXPECTED: 4 | ACTUAL: " + q3.size());
    }
}
```