**CS 1450 Data Structures and Algorithms – Spring 2020**

**Assignment #7**

Due Date: **MONDAY March 30, 2020** at 1:40pm (MW class), **TUESDAY** **March 31, 2020** at 1:40pm (TR class)

Purpose: Learn about regular queues and priority queues.

Effort: Individual

Points: **100**

Deliverables: Upload a **.zip** file with **ONLY** your source code (one .java file) to Canvas.

**Hand in a hardcopy version of the code and your design notes at beginning of class.**

**The Situation**

In this assignment, you are going to simulate a single-player escape room game for the top ranked players in the world. When the players arrive, they are placed into designated *seats* outside the escape game. Next, the players join the escape game *waiting queue* and wait for their turn to enter the actual escape room where the game is played. Each player will be removed from the *waiting queue*, enter the *escape room*, obtain a score, and then be moved to the *results queue*, where their position depends on the score they received in the escape room.

**Assignment Description**

This assignment provides the opportunity to learn how to use regular queues, to implement and use priority queues, and work with queues nested within a class. It also provides the opportunity to implement a hash function. The completed program contains these classes:

* **Player:** contains details for a player
* **EscapeRoom:** this is the room where the escape game is played (simulates player trying to escape)
* **EscapeGame:** contains one *waiting line* (queue of players)*,* one *results queue* (priority queue of players) and one escape room.
* **EscapeGameController:** controls movement of players from *waiting line* to *escape room* to *results line*.
* **PriorityQueue:** contains array of players and all methods to make that array represent a priority queue.

**Specifications**

1. Create a Java class called **LastNameFirstNameAssignment7**
2. Follow "CS1450 Programming Assignments Policy"
3. Write a test program (i.e. main) to do the following:
   1. Setup steps
      1. Create an array to hold players (set size to 25)
         * This array represents the players waiting outside the game room – call it **seats.**
         * It is not part of any class so create it in main.
      2. Read players from the file **players.txt**
      3. Create players and add to the *seats* array in the correct location
      4. Create an **escape game** object and an **escape game controller** object
   2. Start escape game simulation
      1. Step 1: **escape game controller** moves all players into the escape game:
         * Move players waiting outside in the *seats* array into the escape game.
         * Print “*Moved into escape game:*” message for each player
         * Use controller’s **movePlayersIntoEscapeGame** method
      2. Step 2: **escape game controller** – simulate player playing escape game
         * Move players one at a time out of the *waiting queue* into the *escape room* to simulate playing the escape game.
         * Print player’s name, the score received, as well as the player that is the current leader
         * Use controller’s **simulateEscapeGame** method
      3. Step 3: **escape game controller** – display the overall results for all players of the game
         * After all players have played, display all players in order from highest to lowest score
         * Use the controller’s **displayResults** method
4. Test file information:
   1. Run your code on the file **players.txt.** This is a test file, do NOT ASSUME that the order or number of players will be the same in the file used for grading.
   2. File layout is:

Player Ranking Seat Number

**Bella 54 33**

**Classes**

**Player**

* Description
  + Represents one player.
  + Must implement **Comparable** so players can be sorted in the *resultsQueue* (EscapeGame’s priority queue)
* Private Data Fields
  + **name** - string for player’s name
  + **ranking** - integer for player’s national ranking
  + **seat** - integer for the seat the player is placed in outside the escape game
  + **score** - integer for the score the player obtains when trying to escape from the escape room
* Public Methods
  + Constructor ***public Player(String name, int ranking, int seat)***
    - Initializes name, ranking, and seat to incoming values
    - Initializes score to 0
  + Getters: For data fields **name, ranking, seat, score**
  + Setters: For data field: **score**
  + ***public int compareTo(Player otherPlayer) -*** compares two players based on their score, returns:
    - -1 if this player’s score is less than passed in player’s score (otherPlayer)
    - 0 if the players have the same score
    - 1 if this player’s score is more than passed in player’s score (otherPlayer)

**EscapeRoom**

* Description
  + Represents the room the player enters and attempts to escape from.
  + To simulate the escape game, the escape room contains a method that gives the player a score.
* Private Data Fields
  + None
* Private Methods
  + **private int hash(String key)** – returns an integer hash of the key, the key can be any length
    - For more information on hash functions see the document **Hash Functions FAQ.**
    - This method uses a hash function to create the player’s score
    - Here is the code for the method, copy it into your assignment

// Return a hash of the key. Key can be any length

// Returns an integer >= 0

**private** **int** hash(String key) {

**int** hash = 0;

**for** (**int** i = 0; i < key.length(); i++) {

hash += key.charAt(i);

hash += (hash << 10);

hash ^= (hash >> 6);

}

hash += (hash << 3);

hash ^= (hash >> 11);

hash += (hash << 15);

**return** Math.*abs*(hash);

} // hash

* Public Methods
  + Constructor - none
  + ***public int tryToEscape( String playerName, int playerRanking)* –** simulates player trying to escape from the room. Returns a score between 0 and 100.
    - Construct a key (a string) by concatenating *playerName* and the *playerRanking.*
    - Call the hash function using the key as a parameter.
    - Obtain the score from the hash function.
      * The hash function returns an integer in the range 0 ... 2,147,483,647. Our escape room players are not that good, so use the modulo operator to limit the score to 100. A modulo of 101 will give scores in the range 0...100:

int score = hash( key ) % (101);

**EscapeGame**

* Description
  + Represents the elements that make up the escape game: 1 waiting queue, 1 results queue and 1 escape room object
* Private Data Fields
  + **waitingQueue** - a *regular* queue of players waiting to enter the escape room to play the game.
  + **resultsQueue** - a *priority* queue of players that have completed the escape game.
  + **escapeRoom** - the escape room object where the player plays the game and obtains a score.
* Public Methods
  + Constructor ***public EscapeGame()***
    - Allocate memory for the regular queue, priority queue, and escape room object
  + Getters and Setters: None
  + ***public boolean isWaitingQueueEmpty()* –** returns true when waiting queue is empty
  + ***public void addPlayerToWaitingQueue(Player player)* –** adds (offers) player to waiting queue
  + ***public Player removePlayerFromWaitingQueue() –*** removes player from waiting queue
  + ***public boolean isResultsQueueEmpty()* –** returns true when results queue is empty
  + ***public void addPlayerToResultsQueue(Player player)* –** adds (offers) player to results queue
  + ***public Player removePlayerFromResultsQueue()* –** removes player from results queue
  + ***public Player peekResultsQueue() –*** returns the player with highest score in results queue
  + ***public int tryToEscape(String playerName, int playerRanking)*** - returns score player obtains inside the escape room. This method calls the *tryToEscape* method on the escape room.

**EscapeGameController**

* Description
  + Contains the business logic for moving players.
  + Players are moved from the *seats* outside the escape game into the escape game’s *waiting queue,* from *waiting queue* into the *escape room* to play the game, and finally into the *results queue*.
* Private Date fields
  + None
* Public Methods
  + Constructor: none
  + **public void movePlayerIntoEscapeGame (Player[] seats, EscapeGame escapeGame)**
    - During the setup step, players were placed into a *seats array* to simulate players arriving and waiting outside the Escape Game.
    - Move each player in the *seats array* into the escape game’s *waiting queue.*
    - When moving a player, print a message with player name and seat number.
    - See output below labeled: Controller: Moving players from outside seats into escape game:

Moved to escape game: Geri from outside seat 2

Moved to escape game: Inga from outside seat 3

etc.

* + **public void simulateGame (EscapeGame escapeGame)** 
    - * Simulates each player entering the escape room and playing the escape game.
      * For each player in the escape game’s *waiting queue*:
      * Remove player from *waiting queue*
      * Have player enter the escape game’s *escape room*. Call **tryToEscape** on escape game sending this player in to obtain player’s score.
      * Add player to *results queue* (priority queue you must implement)
      * Print player’s name and score as well as which player is the current leader
        1. See output below labeled - Controller: Starting Escape Game...

------------------------------

Player Score Current Leader

-----------------------------

Geri 60 Geri

Inga 19 Geri

Juliet 26 Geri

Haley 79 Haley

etc.

* + **public void displayResults (EscapeGame escapeGame)** 
    - Display the final standings.
    - Remove each player from the results queue.
    - Display that player’s name and score.

**PriorityQueue**

* Description:
  + You must implement a priority queue from scratch. **DO NOT** use Java’s PriorityQueue class.
  + You will use the implementation you create for the EscapeGame’s *resultsQueue*
  + Represents a priority queue of players, ordered by the score they receive in the escape room.
  + Removing a player from the priority queue returns the player with the highest score.
* Private Data Fields
  + **list** – use an **array** as the storage container (i.e. structure to hold objects in priority queue)
    - To implement a priority queue:
      * Use a basic array of max size 30.
      * Store players in the array based on their scores as follows:
        + The player with the lowest score is at **list[0]**
        + The player with the highest score is at **list[numPlayers-1]**.
      * If there are any available slots after the last player, they will be null, as shown in example below.
    - After a player is added, the array must be sorted to ensure highest score player is at end.
  + **numPlayers** – integer that keeps track of how many players are in the array
    - Since the array will contain players **and** empty locations we need a way to determine how many players are currently in the array.
    - When numPlayers = 0, the array is empty
  + Example of the array:
    - **list** - array with 30 slots (some may contain players, the rest wiil be empty)
    - **numPlayers** - is 3 because the array contains 3 players
    - The last player (highest score player) in the array is always in location **numPlayers-1**
    - The next available slot in the array is always at location **numPlayers**

0 1 2 3 4 29

-------------------------------------------------------------------------

| Player | Player | Player | null | null | ... | null |

-------------------------------------------------------------------------

* Public Methods
  + Constructor: ***public******PriorityQueue()***
    - Allocates memory for the array
    - All slots in the array will initially be empty (null) by default
    - Sets numPlayers to 0
  + ***Getters/Setters:*** *None* (numPlayers is only used by the PriorityQueue class so get/set not needed*)*
  + ***public boolean isEmpty()***
    - Indicates if the array is currently empty (no players are in the array)
  + ***public Player peek()***
    - Returns player with the highest score (last player in array), does NOT remove that player
    - If array is empty, returns null.
  + ***public boolean offer(Player player)***
    - If the array is full, return false.
    - If the array is not full, add the player to the array, sorts the array, and returns true.
    - The next available slot a new player can be added is always at location **numPlayers** .
    - After the player is added, call ***selectionSort*** to sort array. We want to keep the array in sorted order after each add.
  + ***public Player remove()***
    - Removes and returns the player with the highest score (last player in the array)
    - Be sure to set the location the player was in to null.
    - Private methods
  + ***private selectionSort (Player[] list, int numPlayers)***
    - Sorts the array of players from lowest to highest score
    - Uses the selection sort approach (see Lecture #12)
    - This is where the **compareTo** method on the player will be used!

**Must Do and Tips**

**Must Do**

* Use a **regular queue** for the **waitingQueue** – use Java Collections Framework queue
* Use a **priority queue** for the **resultsQueue** – you must write your own priority queue
* Move players out of **seats array** in order from 0 to 24.

**Tip: Sorting and Comparable**

* For the priority queue to work, the Player class must implement Comparable
* The **compareTo** method in the player class will compare player’s based on the score they obtained

**Output**

Your output should look like what is shown below.

Controller: Moving players from outside seats into escape game:

--------------------------------------------------------------------------------

Moved into escape game: Geri from outside seat 2

Moved into escape game: Inga from outside seat 3

Moved into escape game: Juliet from outside seat 8

Moved into escape game: Haley from outside seat 12

Moved into escape game: Bella from outside seat 13

Moved into escape game: Chris from outside seat 17

Moved into escape game: Dell from outside seat 18

Moved into escape game: Frank from outside seat 21

Controller: Starting Escape Game - moving players waiting in line into escape room:

-------------------------------------------------------------------------------------

Player Score Current Leader

-------------------------------------------------------------------------------------

Geri 60 Geri

Inga 19 Geri

Juliet 26 Geri

Haley 79 Haley

Bella 63 Haley

Chris 45 Haley

Dell 88 Dell

Frank 44 Dell

Controller: Escape Room Results

-------------------------------

Player Score

-------------------------------

Dell 88

Haley 79

Bella 63 Players coming out of the results queue

Geri 60 are in order from highest score to lowest

Chris 45

Frank 44

Juliet 26

Inga 19