**Assignment 9 – Queues**

**CSC 300 Fall 2024**

Notes: All homework must be submitted via e-mail. All parts of assignment must be submitted in a single e-mail with multiple attachments when required.

Notes: All homework must be submitted via e-mail. All parts of assignment must be submitted in a single e-mail with multiple attachments. E-mail address is:

[csc300csudh@gmail.com](mailto:csc300csudh@gmail.com)

Each program is to be submitted in a separate file with the file name being the class name with extension .java. I only need the source file. (Only relevant when software is assigned).

The other homework file should be submitted with your name using the following format:

**RosenthalH\_AZ.docx**, where Z is the assignment number. Note: use your name, not mine

**Total Points - 215**

**Short answer questions. (2 points each)**

1. **Determine whether each statement is true or false.**
2. You cannot put null elements into object instantiated from the PriorityQueue class according to the API? \_\_\_\_True\_\_\_\_
3. A slide at the playground is more like a stack than a queue. \_\_\_\_False\_\_\_\_
4. You can instantiate a PriorityQueue using a Dequeue interface reference variable. \_\_\_\_False\_\_\_\_
5. A PriorityQueue is always sorted in *natural order*? \_\_\_\_False\_\_\_\_
6. Within a PriorityQueue the elements sorted last are processed first? \_\_\_\_False\_\_\_\_
7. In a discrete event simulation more than one event can occur at the same instant of time. \_\_\_\_True\_\_\_\_
8. **Multiple Choice**
9. What type of ordering does a PriorityQueue have? \_\_\_C\_\_\_\_\_
10. unordered and unsorted
11. ordered and unsorted
12. ordered and sorted
13. A linear data structure in which insertion and deletion operations can both be performed from both the ends is \_\_\_\_A\_\_\_\_
14. Deque
15. Queue
16. PriorityQueue
17. Circular queue
18. What will be the output of the following program? \_\_\_\_C\_\_\_\_

import java.util.\*;  
public class Book

{

public static void main(String[] args)

{  
    PriorityQueue<String> bk = new PriorityQueue<String>();  
    bk.add("paper");  
    bk.add("pen");  
    bk.add("pencil");  
    System.out.println(bk.peek() + ", " + bk.poll());  
  }  
}

1. paper, pen
2. pencil, paper
3. paper, paper
4. pencil, pencil
5. What will be the output of the following program? \_\_\_A\_\_\_\_\_

import java.util.\*;  
public class Book

{  
    public static void main(String[] args) {  
        PriorityQueue<String> bk = new PriorityQueue<String>();  
        bk.add("paper");  
        bk.add("pen");  
        bk.add("pencil");  
        System.out.println(bk.poll() + "," + bk.peek());  
    }  
}

1. paper, pen
2. pencil, paper
3. paper, paper
4. pencil, pencil
5. **Fill in the blanks**
6. A queue is referred to as a(n) \_\_\_\_\_\_\_FIFO\_\_\_\_\_\_\_\_\_\_\_\_ data structure because the first nodes inserted are the first ones removed.
7. A(n) \_\_\_\_\_Queue\_\_\_\_\_\_\_\_\_ is a constrained version of a LinkedList in which nodes can be inserted only at the end of the list and deleted only from the start of the list.

**Programs**

1. **Print Queue (100)**

In this problem you will simulate multiple printers processing jobs from a single print queue. To do this you will need to create the following classes:

TestPrintQueueSimulation – which includes the main program.

PrintQueueSimulation – controls the simulation and prints the outputs

Job – A class from which printing jobs are created

Printer – an class from which Printers are created

Below is a description to guide you through the program. Remember that object instance variables are initialized to null unless overwritten.

**Job class (20)**

Job has the following variables (You define the access modifier ):

int id;

int arrivalTime;

int timeForJob;

int priority;

private int startTime; //fpr job

int waitTime; //in queue

int endTime; //for job

static int idCounter = 1;

The Job implements the Comparable interface

A Job has a default constructor that does nothing.

A Job has a constructor that is instantiated with three int variables, indicating the arrivalTime, the timeForTheJob, and the priority.

Call setID with no parameters

Then call the set methods for each of the parameters in the constructor

The following methods are required; Please read any special programming required for certain of the set methods. The accessors are all standard with no extra processing

setID and getID,

setArrivalTime and getArrivalTime

setTimeForJob and getTimeForJob

setPriority and getPriority

setStartTime and getStartTime

setEndTime and getEndTime

getWaitTime

Special mutator instructions

When calling **setID**

Set to the idCounter.

Increment the idCounter

When calling **setStartTime** to set the startTime you must also calculate and set the waitTime

The **compareTo** interface is implemented based on priority, with priority 1 Jobs printed before

priority two, etc.

**Printer class (20)**

Printer class has the following variables (You define the access modifier ):

Job printJob; This instance variable is either null or refers to the current Job being processed by the printer.

String printerName;

int startIdleTime;

startInUseTime;

totalIdleTime;

totalInUseTime;

totalJobsProcessed;

Printer()

A Printer class has a default constructor that does nothing

A Printer class has a construcmor that is instantiated with a printerName.

Use the appropriate Mutator method to set the printerName

The following methods are required: - note special instructions

setPrinterName and getPrinterName

setJob and getJob

setStartInUseTime – accepts a time (as an int) and includes incrementing the totalJobsProcessed variable

setStartIdleTime – accepts a time (as an int) and includes updating the totalInUseTime

getTotalIdleTime – easily calculated at the end of the simulation since it is called with currentTime as an input parameter

getTotalInUseTime

getTotalJobsProcessed

**PrintQueueSimulation class (60)**

This is the main driver of this simulation. It is instantiated with the number of printers and the number of jobs to be processed. The simulation runs until all the jobs are processed. There is a single print queue that all printers service. As soon as a printer finishes it get the next job (if any exists) from the waitQueue. Printer Printer0 has priority over Printer1, etc. The simulate method (see below) runs the simulation and calls displayStatistics once the simulation completes.

This class has the following instance variables:

waitQueue which is a PriorityQueue where the references to arriving Jobs are placed, until they pulled by a Printer. (Instantiate as a PriorityQueue based on the natural order which is of course based on the compareTo method in Job)

an int totalWaitTime which keeps track of the total wait time for all Jobs

finishedQueue which is a PriorityQueue where a Job reference is placed when a job is completed

an int currentTime is the simulation clock which **starts at 0** and is incremented 1 second at a time

printer which is a reference to an array of Printer objects

randy – a reference to a Random object

two int variables – numberOfPrinters and numberOfPrintJobs

The PrintQueueSimulation constructor’s parameters are the numberOfJobs, numberOfPrinters, and a seed for the Random object randy.

It uses the first two values to set the instance variables numberOfJobs and numberOfPrinters. (You don’t need to create mutators here)

It creates a Random object randy with the seed in the from the constructor parameter list.

It creates a waitQueue instantiated as a PriorityQueue with a capacity equal to the numberOfPrintJobs

It creates a finishedQueue instantiated as a PriorityQueue with a capacity equal to the numberOfPrintJobs.

It also creates an array printer of length numberOfPrinters.

It then creates a set of printers that total the numberOfprinters, with the array printer referring to each of them in order. As the printer is created the printer names will be Printerl0, Printer1, …

(Hint: printer[i] = new Printer("Printer" +i);

**simulate() method**

This method will simulate the completion of all the print jobs, using the numberOfPrinters as defined.

You may want to create some local variables to help with this method

I created a flagDone which I set to false, allowing me to loop through the logic below until all the Jobs were printed.

For this simulation a new job will be created every 80 seconds. When a Job is created it is randomly assigned a jobTime of between 10 and 1100 seconds inclusive.

It is also assigned a priority of from 1 to 11 inclusive

Below is an outline of the logic, but you will need to do some work and coding

While all jobs aren’t complete (flag is false)

If a job needs to be created (Every 80 seconds (%80 == 0) and not all jobs yet created

Generate a jobTime and jobPriority create a Job with the required 3 parameters

Add the Job to the waitQueue

Increment the jobNumber(just a counter to keep track of number of jobs created)

//See if any Job is fished

For each printer

If there is a Job in the printer

If the Job is complete

Set the endTime for the Job

Place job in the finishedQueue

Set the printer’s Job to null

Set the startIdleTime for the printer to the currentTime

//See if a printer is idle. If so and a job is waiting and assign it to the printer if there is one

For each printer

If the printer is idle (i.e. the Job is null)

If the waitQueue isn’t empty

Remove a Job from the waitQueue

Set the startTime for the Job.

Place the Job in the printer (i.e. use setJob for the printer)

Set the startInUseTime for the printer

Update the totalWaitTime

Increment the currentTime

If the number of jobs to arrive is complete and the waitQueue is empty

Set the flag to true

If there is still a job in any printer

Set the flag back to false

End of loop

**displayStatistics()** (don’t forget the throws Exception)

Create a Scanner to read from the keyboard

Read in the name of an output file and create a PrintWriter based on that name

Note: Print out the Job Statistics table using the finishedQueue which will print the Jobs in order based on theor priotities

Produce a report like the one in the printerstats.txt file in the tab for PROGRAMS/LESSON 9/HOMEWORK.

**TestPrintQueueSimulation class (10)** (don’t forget the throws Exception)

Create a Scanner to read in data via the keyboard

Request the number of printers, the number of print jobs and a seed.

Create a PrintQueueSimulation object

Call the simulate method for the object you have just created

Call the **displayStatistics** method for the object you have just created

**Sample Input**

Please enter the number of printers for the simulation: 6

Please enter the number of printer jobs for the simulation: 45

Please enter a random number seed for the simulation: 2

Enter the name of your output file for the results: printerstats.txt

1. **Airline Boarding (91)**

The following program simulates an airline sales and boarding system. You don’t really need to know anything about airlines to do this problem. The purpose is to sell tickets and then line up and board passengers according to their priorities. I have included a sample output file seating.txt found in the tab Programs/Lesson 9/Homework on the website or in Homework folder for Lesson 9 on Canvass

The six classes are Passenger, FirstClassPassenger, CoachPassenger, PassengerComparator, Flight, and OperateAirline. OperateAirline contains the main method.

In the descriptions below I do not include the access level of the variables and methods, but leave that to you. I also do not provide any of the import statements you may need. I also do not say which method(s) need to have a “throws” statement after the parameter list.

**class Passenger (15)**

class Passenger has the following instance variables:

String passengerID

String ticketClass

int ticketNumber

There is one class variable:

static int ticketCounter initialized to 0

There is a single constructor with a single parameter holding the ticketClass

It call the following three methods:

setTicketNumber()

setTicketClass(ticketClass)

setPassengerID()

class Passenger has the following mutator methods:

**void setTicketNumber()** which increments the ticketCounter and then sets the ticketNumber to the ticketCounter

**void setTicketClass(String tClass)** which sets ticketClass to tClass

**void setPassengerID()** which sets the passengerID to look like the concatenation of “PID\_”, the ticketClass and the ticketNumber: i.e: PID\_Coach 42

It includes accessor methods for each of the three instance variables.

There is a **toString()** method which prints of the passengerID, i.e.:

PID\_First 23

**class FirstClassPassenger is a subclass of Passenger (3)**

Has a single constructor with no parameters

Constructor calls super with the parameter value “First”.

**class CoachPassenger is a subclass of Passenger (3)**

Has a single constructor with no parameters

Constructor calls super with the parameter value “Coach”.

**class PassengerComparator implements Comparator for a Passenger (15)**

Implement a comparator where a FirstClassPassenger precedes a CoachPassenger,. If both Passengers are of the same TicketClass then the Passenger with the lower TicketNumber precedes the Passenger with the higher TicketNumber.

**class Flight (40)**

class Flighthas the following instance variables:

String flightName

int seats

int seatsSold

Random randy

A LinkedList passBooked that contains Passengers

A PriorityQueue boardingQueue of type Passenger that orders based on the PassengerComparator

(Note: passBooked and boardingQueue may be instantiated in the class variables list

There is one constructor:

Flight(String flightName, int seed)

It calls setFlightName(flightName)

It creates an object referenced by randy with the parameter seed as the seed.

It calls setSeats with no parameters

class Flight has the following methods:

**void setFlightName(String fName)** which sets the flightName

**void setSeats()** sets the seats to a number of between 75 and 150 (inclusive) using randy.

**void sellSeats()**

Calculates the seatsSold as the sum of half the number of seats and a random number from 0 to 1 more than half the number of seats, all using integer arithmetic.

i.e.: **randy.nextInt(0, 1 + seats/2) + seats/2**

For each seat to be sold

Randomly select a number from 0 to 4 inclusively

if the number selected is 0

Add a new FirstClassPassenger to passBooked

else

Add a new CoachPassenger to passBooked

**void lineUpCall()**

Use an iterator to add each Passenger in passBooked to boardingQueue

**boarding(PrintWriter outputWriter)**

Create the output similar to that shown in the seating.txt file found in Programs/Lesson 9/Homeworks tab, boarding passengers one at a time based on their position in the boardingQueue. There should be three passengers per row as well as the title line:

**The boarding order for Flight AAA 99 with 102 seats and 51 seats sold is:**

**class OperateAirline (15)**

Below operations all take place in the **main** method

Create an ArrayList flightList of type Flight

Create a Scanner for interactive input

For 3 flights

Request and read in the name of the flight

Request and read in a seed

Create a Flight and add it to flightList

Request and read in the name of the output file

Create a PrintWriter object based on that output file

For each flight in flightList

Call sellSeats

Call lineUpCall

Call boarding with the PrintWriter reference as the parameter.

Have the PrintWriter skip two lines

Close the PrintWriter object

The interactive input may look like this:

Please enter the Flight Name:

AAA 99

Please enter an integer seed from 1 to 50:

5

Please enter the Flight Name:

AAA 128

Please enter an integer seed from 1 to 50:

1

Please enter the Flight Name:

AAA 2020

Please enter an integer seed from 1 to 50:

47

Please enter the name of the output file: seating.txt