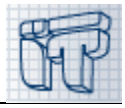


FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second midterm exam

Second Part: Problems (7 points out of 10)

Duration: 90 minutes
Highest score possible: 7 points
Date: May 10, 2019

Overall instructions for the exam:

- Books, notes, mobile phones, as well as other electronic devices are not allowed during the exam. Breaking this rule may result in expulsion from the examination
- Complete your personal information before starting the exam.
- The exam must be filled in with blue or black pen. The final version of the exam must not be filled in with pencil.

Problem 1 (4 / 7 points)

The qualifying of the 2019 Spanish Grand Prix in Montmeló will be held tomorrow. The organizers are facing some issues with the timing application and they have asked you to support them in the last minute. You are required to implement the leaderboard with a linked list. This leaderboard will show the names of the drivers in order, according to their fastest lap (in seconds). In order to do that, you are given a partial implementation of the following classes:

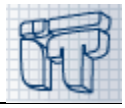
- Node: Stores the name of the driver and his fastest lap. The node also has a reference to the next node.
- Queue<E>: It implements the behavior of a queue. It is already programmed, and it implements the interface QueueInterface<E>
- Leaderboard: It implements the list with the drivers' time. It contains the reference to the first node, a queue to store the lap history (i.e., all the laps in the track) and the size (i.e., number of drivers). Some methods of this class are already implemented.

What follows is the code of the partial implementation of the abovementioned classes. You will be asked to implement some methods, but <u>you cannot implement any additional method</u>.

```
public class Node {
private String driver;
private double fastestLap;
private Node next;
public Node(String driver, double
fastestlap){
    this.driver = driver;
    this.fastestLap = fastestlap;
}

public double getFastestLap(){
    return fastestLap; }
public void setFastestLap(double f){
    fastestLap = f; }
public Node getNext(){
    return next;}
public void setNext(Node next){
    this.next = next; }
public String getDriver(){
    return driver; }
public void setDriver(String dr){
    driver = dr; }
}
```

```
public interface QueueInterface<E> {
        void enqueue(E e);
        E dequeue();
}
```

```
public class Leaderboard {
private Node first;
private Queue<Node> history;
private int size;

public Leaderboard(ArrayList<String> drivers){
    init(drivers);
    history = new Queue<Node>();
    size = drivers.size();
}
public void init(ArrayList<String> drivers){
    // SECTION 1.1
}
public void selectionSort(){
    // SECTION 1.2
}
public void newFastestLap(String driver, double time){
    // SECTION 1.3
}


public Node getNode(int pos){...}
public void swap(int pos1, int pos2){...}
public void print(){...}
}
```



Section 1.1 (1 point)

Implement the method `init(ArrayList<String> drivers)` of class Leaderboard, which is called in the constructor to initialize the list. This method receives an `ArrayList` with the name of the drivers who will participate in the Grand Prix. You have to initialize the linked list with the drivers by inserting them one by one at the beginning of the list (`first` should always point at the last inserted driver). As they do not have completed any lap at the beginning, you can set the fastest lap as the maximum possible time (Double.POSITIVE_INFINITY). After creating the leaderboard, you will have an empty list containing all the drivers with Double.POSITIVE_INFINITY as their time. You can see it in the following example. Note that the word *Infinity* is automatically printed by Java when printing a double with value Double.POSITIVE_INFINITY.

```
public static void main(String []args){
    ArrayList<String> drivers = new ArrayList<String>();
    drivers.add("Lewis Hamilton");
    drivers.add("Sebastian Vettel");
    drivers.add("Valtteri Bottas");
    Leaderboard l = new Leaderboard(drivers);
    l.print();
}
```

```
Valtteri Bottas     Infinity
Sebastian Vettel    Infinity
Lewis Hamilton      Infinity
```

NOTE: Some of the methods of `ArrayList<E>`, which may be useful for this section are:

- `boolean add(E e)`
- `E get(int index)`
- `int indexOf(Object o)`
- `int size()`

Section 1.2 (1.5 points)

Implement the method `selectionSort()`. This method sorts the list of drivers according to their fastest lap using the Selection Sort algorithm. This method should use the version of the Selection Sort algorithm which sorts in ascending order (from lowest time to highest time, that is from fastest driver to slowest driver), searching for the fastest driver in the unsorted part of the array and swapping it with the first unsorted element. You can use the already implemented methods `getNode(int pos)` and `swap(int pos1, int pos2)` to get the node in a specified position, and swap two nodes given their positions, respectively. NOTE: You do not need to implement `getNode(int pos)` and `swap(int pos1, int pos2)`

Section 1.3 (1.5 points)

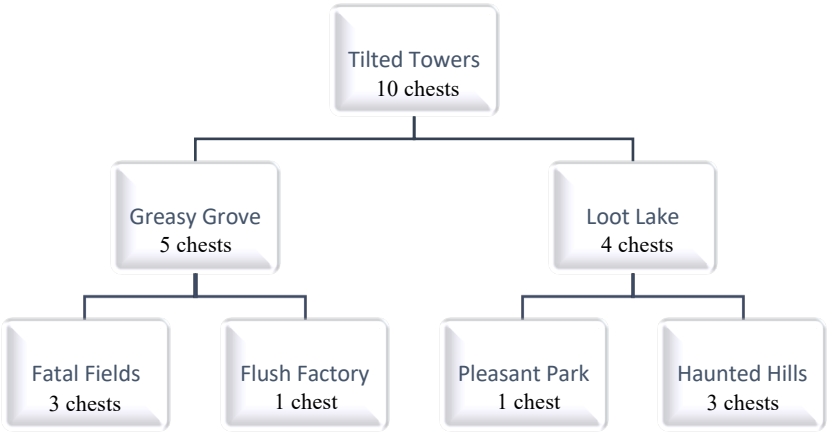
Implement the method `newFastestLap(String driver, double time)`. This method sets the new fastest lap of the driver and updates the classification. In order to do that, the driver is searched in the list. If the driver is found, his new time is compared with his best time (as it is stored in the Node). If the new time is lower than the stored time, the stored time will be updated with the new time. In that case, you also need to internally call the selection sort method programmed in Section 1.2 to update the overall classification. Moreover, you need to (create and) add an independent node with the driver and his time to the history queue which keeps a record with all the laps in the track. You need to add this node no matter if the driver's time is better or not, but only if the drive is found in the list. What follows is an example of calls to this method in the main method:

```
l.newFastestLap("Valtteri Bottas", 100.456);
l.newFastestLap("Lewis Hamilton", 100.324);
l.newFastestLap("Valtteri Bottas", 100.399);
l.newFastestLap("Valtteri Bottas", 100.321);
l.newFastestLap("Sebastian Vettel", 100.399);
l.print();
```

```
Valtteri Bottas    100.321
Lewis Hamilton     100.324
Sebastian Vettel   100.399
```

Problem 2 (3 / 7 points)

A good friend of ours requests our help as engineers to help him to win a game. The game consists of visiting different cities and collecting treasures which are inside chests. Our friend is looking for an algorithm to compute the optimal path along the cities to achieve the maximum number of treasures/chests. For this purpose, a tree data structure has been defined. The root of this tree is the city where the player starts at the beginning of the game and the rest of the nodes represent the cities of the map where the player can go afterwards. An example of the tree can be seen here:



The expected output of the algorithm for this tree is:

Tilted Towers – Greasy Grove – Fatal Fields

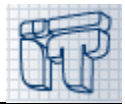
Notice that this path achieves the maximum possible number of chests for this tree: 18.

To model the cities, the class `City` has been implemented. This class has four attributes: (1) `name`, (2) `number_chests`, (3) `option1` and (4) `option2`. Attribute `name` represents the name of the city (e.g., "Tilted Towers"), `number_chests` stores the number of chests/treasures that can be found in the city. Finally, `option1` and `option2` store the subtrees which can be followed.

Moreover, there is a class `CityTree` to model the trees, which only contains the reference to the root. This root is the starting city of the player, which is used to build the tree upon it.

```
public class City {
private String name;
private int number_chests;
private CityTree option1, option2;

public City(String name, int chests){
    this.name = name;
    this.number_chests = chests;
}

public String getName() {
    return name; }
public int getNumber_chests() {
    return number_chests; }

public CityTree getOption1() {
     return option1; }
public void setOption1(CityTree opt1) {
     this.option1 = opt1; }
public CityTree getOption2() {
     return option2; }
public void setOption2(CityTree opt2) {
     this.option2 = opt2; }


}
```

```
public class CityTree {
private City root;

public CityTree(String name, int number_chests){
    this.root = new City(name,number_chests); }

public void insert(CityTree subtree, int option){
    if(root==null) return;
    if(option==1){
        this.root.setOption1(subtree);
    } else if (option==2){
        this.root.setOption2(subtree);
    }
}

public int maximumChests(){
    if(root==null){ return 0; }
    if(root.getOption1()!=null && root.getOption2()!=null){
        return root.getNumber_chests() +
            Math.max(root.getOption1().maximumChests(),
                     root.getOption2().maximumChests());
    } else { return root.getNumber_chests(); }
}

public String computeOptimal(){ //Section 2.2 }
public static void main(String args[]){ //Section 2.1 }

}
```



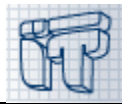
Section 2.1 (1.5 points)

We will have to show our friend how the program works. He knows how to compile and run Java programs, but he does not really understand the Java programming language. For this reason, you need to write a main method which creates the tree shown in the figure above and prints the optimal path (you can use the method `computeOptimal()` as it were already implemented).

Section 2.2 (1.5 points)

Implement the method `computeOptimal()`. This method returns in a String the optimal route (the one with a higher number of chests). You can use the already-implemented method `maximumChests()` which returns the maximum number of chests that can be achieved in a certain tree. You must use recursion in the implementation of the method.

NOTE: Check the implementation of the method `maximumChests()`. Notice that we only consider cities which either have 2 options or 0 options (but they can never have just one option). Assume this fact as true to solve this problem.



REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (1 point)

```
public void init(ArrayList<String> drivers){
	for(int i=0; i<drivers.size(); i++){
		Node newDriver = new Node(drivers.get(i), Double.POSITIVE_INFINITY);
		newDriver.setNext(first);
		first = newDriver;
	}
}
```

Section 1.2 (1.5 points)

```
public void selectionSort(){
	for(int i=0; i<size; i++){
		int m = i;
		for(int j=i; j<size; j++){
			if(getNode(j).getFastestLap()<getNode(m).getFastestLap()){
				m = j;
			}
		}
		swap(i, m);
	}
}
```

Section 1.3 (1.5 points)

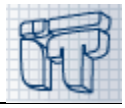
```
public void newFastestLap(String driver, double time){
	Node current = first;
	while(current != null){
		if(current.getDriver().equals(driver)){
			if(time < current.getFastestLap()){
				current.setFastestLap(time);
				selectionSort();
			}
			history.enqueue(new Node(driver, time));
			break;
		}
		current = current.getNext();
	}
}
```

Section 1.1 (1 point)

- 0.25: Loop to consider all the elements of the ArrayList
- 0.75: Insertion of the node at the beginning of the list
  - 0.25: Creation of the node
  - 0.25: Link node and first
  - 0.25: Update first
- Significant errors are subject to additional penalties

Section 1.2 (1.5 points)

- 0.25: External loop, from 0 to size or size – 1
- 0.25: Initialize m
- 0.25: Internal loop from i to size
- 0.25: Comparison of times in ascending order. If the comparison is correct, but not the order, maximum 0.15.
- 0.25: Get the minimum value into a local variable



- 0.25: Calling the method that swaps correctly
- Significant errors are subject to additional penalties

Section 1.3 (1.5 points)

- 0.25: Traverse the list with an auxiliary node
- 0.25: Check if the driver given as a parameter is the same of the driver in current node
- 0.25: Check if the lap is faster than the current fastest lap of the driver
- 0.25: Update fastest lap when new lap is faster
- 0.25: Call to selection sort when the fastest lap has changed. If selection sort is always called (it is called outside the if), penalize 0.1.
- 0.25: Enqueue a node with the driver and time. If the enqueue call is carried out only when the lap is faster than the current lap or the call is carried out even when the driver is not found, penalize 0.1. Penalize 0.1 current is enqueued instead of a new node.
- Significant errors are subject to additional penalties

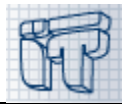
PROBLEM 2

Section 2.1 (1.5 points)

```
public static void main (String args[]){
	CityTree t1 = new CityTree("Tilted Towers",10);
	CityTree t2 = new CityTree("Greasy Grove",5);
	CityTree t3 = new CityTree("Loot Lake",4);
	CityTree t4 = new CityTree("Fatal Fields",3);
	CityTree t5 = new CityTree("Flush Factory",1);
	CityTree t6 = new CityTree("Pleasant Park",1);
	CityTree t7 = new CityTree("Haunted Hills",3);
	t2.insert(t4, 1);
	t2.insert(t5, 2);
	t3.insert(t6, 1);
	t3.insert(t7, 2);
	t1.insert(t2, 1);
	t1.insert(t3, 2);
	System.out.println(l1.computeOptimal());

}
```

Section 2.2 (1.5 points)

```
public String computeOptimal(){
	if(this.root==null){
		return "";
	}
	if(this.root.getOption1()!=null && this.root.getOption2()!=null){
		if(root.getOption1().maximumChests() > root.getOption2().maximumChests()){
			return this.root.getName() + root.getOption1().computeOptimal();
		} else{
			return this.root.getName() + root.getOption2().computeOptimal();
		}
	} else{
		return this.root.getName();
	}

}
```



Section 2.1 (1.5 points)

- 0.6: Creation of the trees.
- 0.6: Insertion of the subtrees.
- 0.3: Printing the optimal route invoking the computeOptimal( ) method
- Significant errors are subject to additional penalties

Section 2.2 (1.5 points)

- 0.25: Checking if the tree is empty (root==null)
- 1.0: Recursive case
    - 0.5: Comparison of the two options using the maximumChests( ) method.
    - 0.5: Recursive call
- 0.25: Base case
    - 0.10: Checking that it is a leaf node (no options).
    - 0.15: Returning the name of the leaf node.
- Significant errors are subject to additional penalties