FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second midterm exam

Second Part: Problems (7 points out of 10)

Duration: 70 minutes
Highest score possible: 7 points
Date: May 13, 2021

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.

Problem (7 points)

A leading logistics company has asked you to develop a program for managing a warehouse. The classes for this program are the following (assume all getters and setters are implemented for these classes):

- Order (`Order`). It represents the document (purchase order) that contains each of the orders that are issued in the warehouse. Each purchase has an order identifier (`orderID`) of type `integer` and the total benefit (`totalBenefit`), of type `double`. All other attributes are not relevant to this exercise.

- Store Manager (`StoreManager`). It is the brain of the application and will contain all the logic of the program. It has a queue with orders (`LinkedQueue<Order>`), which stores the orders (`Order`) to be processed (`ordersToProcess`), a binary search tree (`LBSTree<Order>`) of orders (`Order`), which stores the orders that have already been processed (`storeOrders`) and a list (`LinkedList<Order>`) of orders (`Order`), which stores the orders which have a high benefit (`highBenefitOrders`).

Section 1.1 (1 point)

Make the class `Order` implement interface `Comparable`. You must write the code for the method `compareTo`. Orders are compared numerically based on their `ID`. Bear in mind that the method `compareTo()` of the interface `Comparable` receives as a parameter an `Object` type. Therefore, inside the method you will have to make an explicit casting to `Order`. If that casting cannot be done, then a `ClassCastException` will be thrown. You do not need to program that exception as it already exists in the Java libraries, but you will have to catch it within the method itself and send a message indicating that the parameter of the method `compareTo()` should be an order (`Order`).

Section 1.2 (1 point)

Write the code for the method `public int indexOf(Order o)` in class `StoreManager` which must search the order received as parameter in the queue `ordersToProcess` which stores the orders and returns the position in the queue where the product is found. In case there is no order with that identifier it returns -1. The queue must remain unaltered when the method finishes.

The queue implements the following interface:

```
public interface Queue<E> {
    boolean isEmpty();
    int size();
    void enqueue (E info);
    E dequeue();
    E front();
}
```

Note: To deal with the queue you must **exclusively** use the methods included in the aforementioned interface. Solutions which do not meet this requirement will not be allowed.

.

Section 1.3 (1,5 points)

Write the code for the method `public void showBenefit()` in class `LBSTree` which prints to output the ID and the total benefit for each order in the tree. You can use the getters in the class `Order`, as they are already implemented. The class `LBSTree` implements the following interface:

```
public interface BSTree<E>
{
       public boolean isEmpty();
       public E getInfo();
       public Comparable getKey();
       public BSTree<E> getLeft();
       public BSTree<E> getRight();
       public void insert (Comparable key, E info);
}
```

Section 1.4 (1,5 points)

Write the code for the method `public void showBenefit()` in class `StoreManager` which must go through all orders in the queue `ordersToProcess` and store them in the tree `storeOrders`, and thereafter it must show the benefit of the orders in the tree. You can use the method in the previous section even though you were not able to complete it.

Note: The queue must remain unaltered when the method finishes.

Section 1.5 (2 points)

Write the code for the method `public void processHighBenefitOrders()` in class `StoreManager` which must go through all orders in the queue `ordersToProcess` and store them in the list `highBenefitOrders`. The method must first compute the benefit average of the orders contained in the queue `ordersToProcess` and then go through the queue `ordersToProcess` again and insert in the list `highBenefitOrders` the orders which have a higher benefit than the average. You can use the method `public void insert (E info)` of class `LinkedList` to add elements to the list. Again, you must use the methods declared in the interface `Queue<E>` and keep the queue unaltered when the method finishes.

REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM

Section 1.1 (1 point)

```
public int compareTo(Object o) throws ClassCastException {
	Order order;
	try {
		order = (Order) o;
	} catch (ClassCastException e) {
		throw new ClassCastException("Bad order argument");
	}

	int result;

	if (this.getID() == order.getID())
		result = 0;
	else if (this.getID() < order.getID())
		result = -1;
	else
		result = 1;

	return result;
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,25 if casting is OK in method compareTo.
- 0,25 if exception capture and launch is correct.
- 0,1 for each correct comparison and return value assignment in method compareTo (0,3 in total)
- 0,2 for returning the correct value in method compareTo.
- Significant errors are subject to additional penalties

Section 1.2 (1 point)

```
public int indexOf(Order o){

		int position = -1;

		for(int i=0;i<ordersToProcess.size(); i++){

			Order auxOrder = ordersToProcess.dequeue();

			if (auxOrder.equals(o))  // also: if (auxOrder.compareTo(o) == 0),
			                         // but the exception must be handled
				position=i;

			ordersToProcess.enqueue(auxOrder);

		}

		return position;
```

```
        }
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,25 if the queue is gone through correctly.
- 0,25 if method equals is used to compare.
- 0,25 if the order is inserted in the queue again.
- 0,25 if the correct result is returned.
- If the interface methods are not used the highest mark is 0,5
- Significant errors are subject to additional penalties

Section 1.3 (1,5 points)

```
public void showBenefit() {
                if (!this.isEmpty()) {

                        if (!this.getLeft().isEmpty())
                                this.getLeft().showBenefit();

                        Order order = (Order) this.getInfo();
                        System.out.println("Order ID: " + order.getID() + " Benefit: " +
order.getTotalBenefit());

                        if (!this.getRight().isEmpty())
                                this.getRight().showBenefit();

                }
        }
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,5 for each recursive call.
- 0,25 if the stopping condition is OK.
- 0,25 if printing to console is OK.
- Significant errors are subject to additional penalties.

Section 1.4 (1,5 points)

```
public void showBenefit(){


                for(int i=0;i<ordersToProcess.size(); i++){
                        Order order = ordersToProcess.dequeue();
                        storeOrders.insert(order.getID(), order);
                        ordersToProcess.enqueue(order);
                }

                storeOrders.showBenefit();
        }
```

Evaluation criteria

- 0 if the code makes no sense.

- 0,2 if the queue is gone through correctly.
- 0,5 if the order is inserted correctly in the tree.
- 0,5 if method showBenefit is called after inserting all orders.
- 0,3 if the order is inserted in the queue again.
- If the interfaces methods are not used the highest mark is 0,75
- Significant errors are subject to additional penalties.

Section 1.5 (2 points)

```
 public void processHighBenefitOrders() {

    double totalBenefit=0, average=0;

    for (int i = 0; i < ordersToProcess.size(); i++) {
       Order order = ordersToProcess.dequeue();
       totalBenefit+=order.calculateTotalBenefit();
       ordersToProcess.enqueue(order);
    }

    if (ordersToProcess.size()>0)
       average = totalBenefit/ordersToProcess.size();

    for (int i = 0; i < ordersToProcess.size(); i++) {
       Order order = ordersToProcess.dequeue();
       if (order.calculateTotalBenefit() > average)
          highBenefitOrders.insert(order);
       ordersToProcess.enqueue(order);
    }
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,5 if the queue is gone through correctly (0,25 for each loop)
- 0,5 if the average is computed correctly (maximum 0,25 if zero size is not checked).
- 0,5 if the original queue is restored (0,25 for each loop)
- 0,5 if the orders are inserted in the list correctly.
- If the interfaces methods are not used the highest mark is 0,75.
- Significant errors are subject to additional penalties.