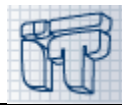


FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second midterm exam

Second Part: Problems (7 points out of 10)

Duration: 80 minutes
Highest possible score: 7 points
Date: May 6, 2022

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 1 (3 / 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 already implemented for all classes):

- Product (`Product`). A type of good that can be distributed from the warehouse. It has a name (`name`) and brand (`brand`). Both attributes are of type `String`. It also has additional information consisting of: an integer number indicating the number of units of that product available in the warehouse (`numUnits`), and two decimal numbers to indicate both the cost of manufacture of the product in euros (`costPerUnit`) and the selling price also in euros (`pricePerUnit`).

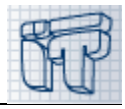
- Person (`Person`). It refers to any relevant person for the management of a warehouse. It represents the customers of the warehouse. The attributes are not relevant to this exercise.

- Order (`Order`). It represents the document (purchase order) that contains each of the orders that are issued in the warehouse. It is a data structure that has a list of products that is declared as `ArrayList<Product> list`. It also has a reference to the client (`client`). This attribute belongs to class `Person`.

- Store Manager (`StoreManager`). It is the brain of the application and will contain all the logic of the program. In this problem the only relevant attribute is a queue with orders (`LinkedQueue<Order>`), which stores the orders to be processed (`ordersToProcess`).

Section 1.1 (1 point)

Make the class `Product` implement the interface `Comparable`. You must write the code for the method `compareTo` in class `Product` Products are alphabetically compared based on their `name` and `brand`. First the names will be compared, and if they are equal, the brand will be compared. 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 `Product`. 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 a product (`Product`).



Remember that the class `String` also implements the interface `Comparable`.

In case the object to compare to is `null`, the method `compareTo` should throw a `NullPointerException`. The method `compareTo` returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the object passed as an argument.

Section 1.2 (1 point)

Write the code for the method `public int countOrders(Person p)` in class `storeManager` which must return the number of orders which are pending to be processed in the queue `ordersToProcess` belonging to the client received as an argument. 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.

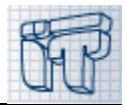
Note: Some methods in class ArrayList<E> that may be useful here are:

- `boolean add(E e)`
- `void add(int index, E element)`
- `void clear()`
- `E get(int index)`
- `int indexOf(Object o)`
- `boolean isEmpty()`
- `E remove(int index)`
- `boolean remove(Object o)`
- `E set(int index, E element)`
- `int size()`

Section 1.3 (1 point)

Write the code for the method `public double getBenefit()` in class `StoreManager` which must go through all orders in the queue `ordersToProcess` and return the total benefit of the orders in the queue. For each order you must get all the products and compute the benefit of each product. The benefit is calculated as the difference between the price and the cost multiplied by the number of units.

Note: As in the previous section, you must use the methods included in the interface and the queue must remain unaltered when the method finishes.



Problem 2 (4 points)

For the development of the problem, classes `LBSNode<E>`, `BinaryTree<E>` and `BinaryTreeExample` are given. These classes allow modeling a binary search tree.

Nota: Exceptions will not be taken into account in any case, and getters and setters are implemented.

```
public class LBSNode <E> {

 private int key;
 private LBSNode<E> left,
 right;

public LBSNode(int data){
 key = data;
 left = right = null; }
}
```

```
public class BinaryTree<E> {

 private LBSNode<E> root;

 public BinaryTree() {
   this.root = null;}

 public boolean isBSTOrNot() {
  return isBSTOrNot(this.root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

public void  insert(int key){…}

public boolean isBSTOrNot(LBSNode<E> root, int minValue, int maxValue)

public LBSNode<E> search_Recursive(LBSNode<E> root, int key)

public void inorder_Recursive(LBSNode<E> root)
```

```
public class BinaryTreeExample {
   public static void main(String args[]) {


   }
```

Section 2.1 (1 point)

Write the code for the method `isBSTOrNot`, this method to check the given tree is Binary search tree or not

Note: A binary search tree (BST) is a node based binary tree data structure which has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Section 2.2 (1 point)

Write the code for the method `search_Recursive`, this method recursively searches for the key in a binary search tree. This method must be implemented recursively; any other implementation will not be scored.

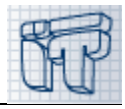
Section 2.3 (0,75 point)

Write the code for the method `inorder_Recursive` this method prints the information contained in the tree nodes following the inorder traversal. This method must be implemented recursively; any other implementation will not be scored.

Section 2.4 (1,25 point)

Write the code for the main program, this must include:

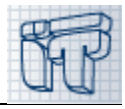
1.- Construction of the next tree



```
/* BST tree example
            100
          /     \
        80       110
       /  \
     79   95    */
```

2.- Check if `isBSTOrNot` an write in the screen the result

3.- Search if the tree is the number **95**



REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (1 point)

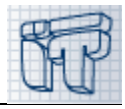
```
public int compareTo(Object o) throws ClassCastException, NullPointerException {

        int result = 0;

        if (o == null)
                throw new NullPointerException("null object");

        Product product;
        try {
                product = (Product) o;
        } catch (ClassCastException e) {
                throw new ClassCastException("Bad product argument");
        }

        // First compare the names
        result = this.getName().compareTo(product.getName());

        if (result == 0)
                // Now compare the brands
                result = this.getBrand().compareTo(product.getBrand());

        return result;

}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 if the signature of the method is correct (throws the exceptions).
- 0.1 if the NullPointerException is thrown.
- 0.15 if casting is OK in method compareTo.
- 0.1 if the exception capture and launch is correct.
- 0.2 if the names are compared correctly.
- 0.2 if the brands are compared correctly.
- 0.05 for each correct return value (equals, lesser, greater).
- Significant errors are subject to additional penalties.

Section 1.2 (1 point)

```
public int countOrders(Person p){

        int result = 0;

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

                Order order = ordersToProcess.dequeue();

                if (order.getClient().equals(p))
                        result++;

                        ordersToProcess.enqueue(order);

        }

        return result;
}
```



Evaluation criteria

- 0 if the code makes no sense.
- 0.3 if the queue is traversed correctly.
- 0.25 if the method equals is used to compare.
- 0.25 if the queue remains unaltered at the end of the method (the element can be enqueued or an auxiliary queue can be used).
- 0.2 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 point)

```
public double getBenefit(){

        double result =0;

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

                for(int j=0;j<order.getList().size();j++){

                        Product p = order.getList().get(j);
                        result += p.getNumUnits()*(p.getPricePerUnit()-p.getCostPerUnit());

                }
                ordersToProcess.enqueue(order);
        }
        return result;
}
```

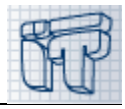
Evaluation criteria

- 0 if the code makes no sense.
- 0,2 if the queue is traversed correctly.
- 0,25 if the list of products is traversed correctly.
- 0,2 if the benefit is updated correctly.
- 0.25 if the queue remains unaltered at the end of the method (the element can be enqueued or an auxiliary queue can be used).
- 0.1 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.

PROBLEM 2

Section 2.1 (1 point)

```
public boolean isBSTOrNot(LBSNode<E> root, int minValue, int maxValue) {
        // check for root is not null or not
        if (root == null) {
            return true;
        }
         // check for current node value with left node value and right node value
         and recursively check for left sub tree and right sub tree
         if(root.getKey() >= minValue && root.getKey() <= maxValue &&
         isBSTOrNot(root.getLeft(), minValue, root.getKey()) && isBSTOrNot(root.getRight(),
         root.getKey(), maxValue)){
            return true;
        }
        return false;
    }
```



Evaluation criteria

- 0 if the code makes no sense.
- 0.1 if the signature of the method is correct.
- 0.4 if check OK the base case
- 0.1 if check OK `root.getKey() >= minValue && root.getKey() <= maxValue`
- 0.2 if check OK `root.left`
- 0.2 if check OK `root.right`
- Significant errors are subject to additional penalties.

Section 2.2 (1 point)

```
public LBSNode<E> search_Recursive(LBSNode root, int key) {
        // Base Cases: root is null or key is present at root
        if (root==null || root.getKey()==key)
            return root;
        // val is greater than root's key
        if (root.getKey() > key)
            return search_Recursive(root.getLeft(), key);
        // val is less than root's key
        return search_Recursive(root.getRight(), key);
}
```

Evaluation criteria

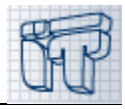
- 0 if the code makes no sense.
- 0.3 if the basic case is OK
    - 0,15. root == null
    - 0,15 root.key == key
- 0.35 if the search is OK for one of the branches
- 0.35 if the search is OK for the other branch
- Significant errors are subject to additional penalties.

Section 2.3 (0.75 points)

```
private void inorder_Recursive(LBSNode root) {
        if (root != null) {
            inorder_Recursive(root.getLeft());
            System.out.print(root.getKey() + " ");
            inorder_Recursive(root.getRight());
        }
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,15 if checks the root ! = null.
- 0,3 if call recursive one branch.
- 0,3 if call recursive the other branch.
- Significant errors are subject to additional penalties.



Section 2.4 (1.25 points)

```
public class BinaryTreeExample {

    public static void main(String[] args) {
        // Creating the object of BinaryTree class
        BinaryTree<Integer> bt = new BinaryTree<Integer>();
        bt.insert(100);
        bt.insert(110);
        bt.insert(80);
        bt.insert(95);
        bt.insert(79);

        System.out.println(bt.isBSTOrNot());

        LBSNode<Integer> find = bt.search_Recursive(bt.getroot(), 95);

        if (find!= null)
             System.out.println("Find it!!");
        else
             System.out.println("Not Find it!!!");
    }
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,1 if BinaryTree() is correctly created
- 0,5 if Tree is correctly created using insert method
- 0,3 if `isBSTOrNot()` is check and call correctly
- 0,35 if the number 95 in the tree is correctly searched
- Significant errors are subject to additional penalties.