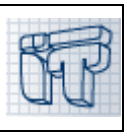
uc3m	Universidad Carlos III de Madrid Department of Telematic Engineering Systems Programming: English Group	

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 11, 2023

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)

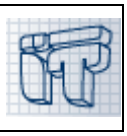
We have the following declarations:

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


public interface Stack<E> {
    boolean isEmpty();
    int size();
    void push (E info);
    E pop();
    E top();
}


public class LinkedQueue<E> implements Queue<E>
{ … }

public class LinkedStack<E> implements Stack<E>
{ … }
```

```
public class Point {
    private double x;
    private double y;

    public double getX()
      { … }

    public double getY();
      { … }

    …
}
```

Assume that all methods in the classes `LinkedQueue` and `LinkedStack` are already implemented. The class `Point` represents geographical points used by a GPS. The coordinate `X` represents the longitude, and the coordinate `Y` represents the latitude of the GPS point.

Section 1.1 (1 point)



You are asked to write the code for the method

```
public LinkedStack<Point> northenHemisphere(LinkedQueue<Point> q)
```

This method receives as parameter a queue which contains elements of type `Point`. The method should return a `LinkStack` with the points which belong to the Northern Hemisphere, i.e., the points which have positive latitude.

VERY IMPORTANT: To deal with the queue and the stack you must **exclusively** use the methods included in the aforementioned interfaces. Solutions which do not meet this requirement will not be allowed. Also the queue must remain unaltered when the method finishes.

Section 1.2 (2 points)

Write the code for the method

```
public boolean foundInBoth(LinkedQueue<Object> q, LinkedStack<Object> s, Object o)
```

This method receives a `Linkedqueue` and a `LinkedStack`, as well as a generic `Object`. The method should return `true` if the object is present in both structures, and `false` otherwise.

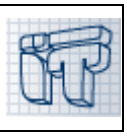
VERY IMPORTANT: As in the previous section, you must **exclusively** use the methods included in the interfaces and both the queue and the stack must remain unaltered when the method finishes.

Problem 2 (1 / 7 points)

Given the class `ArrayList<E>` with the following methods:

```
public E get(int index)

public E set(int index, E element)

public int size()
```

Implement the method `public void SelectionSort(ArrayList<Integer> list)`. The method receives an object `ArrayList` containing objects of type `Integer` and it must sort the elements in ascending order. The sorting method to be implemented is `Selection Sort`.



Problem 3 (3 / 7 points)

A leading logistics company has asked you to develop a program for managing a Warehouse of products located in the Iowa offices.

The Product (`Product`) class, which is already implemented and will be used in this case, is a tangible item that is offered for sale and it can be distributed from the Warehouse. The attributes of `Product` is:

- Code (code): The code of the product. This attribute is the key to sorting the products in the warehouse management.
- Name (name). Name of the product.
- Price (`price`). Price of the product.
- Category (`category`). Category of the product.

Assume all `getters`,`setters` and the method `compareTo`, are already implemented in the class Product.

You must implement a class named `WarehouseManagement`. This class is the brain of the application and will contain the basic operations for warehouse management (`addProduct`, `findProduct`). To manage the Products, you should use a binary search tree (LBSTree) as the data structure whose interface is `BSTree`.

```
public interface BSTree<E> {

   boolean isEmpty();

   E getInfo();

   Comparable getKey();

   BSTree<E> getLeft();

   BSTree<E> getRight();

   void insert(Comparable key, E info);

   BSTree<E> search(Comparable key);}
```

Section 1 (0.5 points)

Define the attributes and implement the constructor without parameters for the class `WarehouseManagement`.

Section 2 (0.5 points)

Write the code to define the basic operations to `addProduct` in `WarehouseManagement`.

```
public void addProduct(int code, Product product){…}
```

Section 3 (0.5 points)

Write the code to define the basic operations `findProduct` in `WarehouseManagement`.

```
public Product findProduct(int code){…}
```

Section 4 (1.5 points)

Write the code of the method `countProducts` in `WarehouseManagement`. This method recursively counts the number of products in the warehouse

```
public int countProducts(BSTree<Product> tree) {…}
```

VERY IMPORTANT: The method **must be implemented recursively**. Any other implementation will not be valid

REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (1 point)

```
public LinkedStack<Point> northenHemisphere(LinkedQueue<Point> q) {

		LinkedStack<Point> s = new LinkedStack<Point>();

		for (int i = 0; i < q.size(); i++) {
			Point p = q.dequeue();

			if (p.getY() > 0)
				s.push(p);

			q.enqueue(p);
		}

		return s;
	}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 if the LinkedStack is correctly declared and initialized.
- 0.3 if the queue is correctly traversed.
- 0.1 if the condition to check the latitude is OK.
- 0.2 if the element is correctly pushed into the stack.
- 0.2 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.

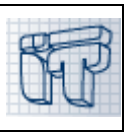
- If the interface methods are not used the highest mark is 0.5.
- Significant errors are subject to additional penalties.

Section 1.2 (2 points)

```
public boolean foundInBoth(LinkedQueue<Object> q, LinkedStack<Object> s,
              Object o) {
       boolean foundInQ = false;
       boolean foundInS = false;
       LinkedStack<Object> aux = new LinkedStack<Object>();

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

              Object obj = q.dequeue();
              if (obj.equals(o))
                     foundInQ = true;
              q.enqueue(obj);
       }

       if (foundInQ)
              for (int i = 0; i < s.size(); i++) {
                     Object obj = s.pop();
                     if (obj.equals(o))
                            foundInS = true;

                     aux.push(obj);
              }

       for (int i = 0; i < aux.size(); i++)
              s.push(aux.pop());

       return foundInQ && foundInS;
       }
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 if the auxiliary LinkedStack is correctly declared and initialized.
- 0.3 if the queue is correctly traversed.
- 0.3 if the stack is correctly traversed.
- 0.3 if the method equals is used to compare.



- 0.2 if the elements are correctly pushed and popped from/to the stack.
- 0.5 if both the queue and the stack remain unaltered at the end of the method (the element can be enqueued or an auxiliary queue can be used. For the stack an auxiliary stack is needed). 0.25 for each correct structure at the end of the method.
- 0.3 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 (1 point)

```
public void SelectionSort(ArrayList<Integer> list) {

              for (int i = 0; i < list.size() - 1; i++) {
                     int m = i;
                     for (int j = i; j < list.size(); j++) {
                            if (list.get(j) < list.get(m) ) {
                                   m = j;
                            }
                     }
                  Integer aux = list.get(i);
                  list.set(i, list.get(m));
                  list.set(m, aux);
              }
       }
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.25 if the loops are OK.
- 0.25 if the comparison is OK.
- 0.5 if the swapping process is OK.

PROBLEM 3 (3 points)

Section 1 (0.5 points)

```
public class WarehouseManagement {
  private BSTree<Product> products;
  public WarehouseManagement() {
         products = new LBSTree<Product>();}
  }
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 if the LBTree is correctly declared and initialised.
- 0.3 if the constructor is OK.
- 0.1 if the variable is correctly assigned.
- Significant errors are subject to additional penalties.

Section 2 (0.5 points)

```
public void addProduct(int code, Product product) {
    if (product != null) {
        products.insert(code, product);
    }

}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0,25 If control that product null
- 0.25 If the call to the function insert in the method addProduct is correct.
- Significant errors are subject to additional penalties

Section 3 (0.5 points)

```
public Product findProduct(int code) {
    BSTree<Product> productTree = products.search(code);
    if (productTree == null) {
        return null;
    }
    return productTree.getInfo();
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.25 If the call to the function search in the method findProduct is correct.
- 0.25 If you return the Product in the method findProduct is correct.
- Significant errors are subject to additional penalties

Section 3 (1.5 point)

```
public int countProducts(BSTree<Product> tree) {
      if (tree.isEmpty()) {
            return 0;
       }
       else {
            int leftCount = countProducts(tree.getLeft());
            int rightCount = countProducts(tree.getRight());
```

```
            return leftCount + rightCount + 1;
        }
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0 if the code is no implemented recursively
- 0.25 If the basic case is correct.
- 0.5. If the recursive call to the right subtree is correct
- 0.5. If the recursive call to the left subtree is correct
- 0.25. If add 1 to the count
- Significant errors are subject to additional penalties.