FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second midterm exam

Second Part: Problems (7 points out of 10)

Duration: 80 minutes
Highest score possible: 7 points
Date: May 12, 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 1 (3 / 7 points)

In the store of the course project (`StoreManager`), there is a queue to store the orders to be processed (`ordersToProcess`). The queue is implemented in another class (`LinkedQueue`), whose main methods are provided below. In the queue, orders are of type `Order`, and are identified with the attribute `orderID`, which can be accessed with its corresponding `getOrderID()`.

```
public class LinkedQueue {
    private Node<Order> head;
    private Node<Order> tail;
    private int size;

    public LinkedQueue() {}
    public boolean isEmpty() {...}
    public int size() {...}
    public Order front() {...}
    public void enqueue(Order info) {...}
    public Order dequeue() {...}
    public String toString() {...}
    public void print() {...}
}
```

Section 1.1 (1.5 points)

Considering the abovementioned information, implement the method `Order search(int orderID)` in class `StoreManager`. This method searches one order in the queue (given its `orderID`) and returns the corresponding `Order` object. For this method, you can only use the methods of class `LinkedQueue` that are already provided. Note that this method does not modify neither the content of the queue nor the order of its elements.

Section 1.2 (1 point)

The implementation of previous section would have been easier if there was a method to search in the `LinkedQueue` class. In this section, you are asked to implement the method `Order search(int orderID)` in `LinkedQueue` class. This method searches one order in the queue (given its `orderID`) and returns the corresponding `Order` object.

Section 1.3 (0.5 points)

Finally, implement the method `Order search2(int orderID)` in `StoreManager`. This method carries out the same operations of the method implemented in Section 1.1, but making use of the method implemented in Section 1.2.

Problem 2 (1 / 7 points)

In a project, we need a stack data structure, and we want to make use of ArrayLists for its implementation. Complete the code below to implement the main methods of the stack.

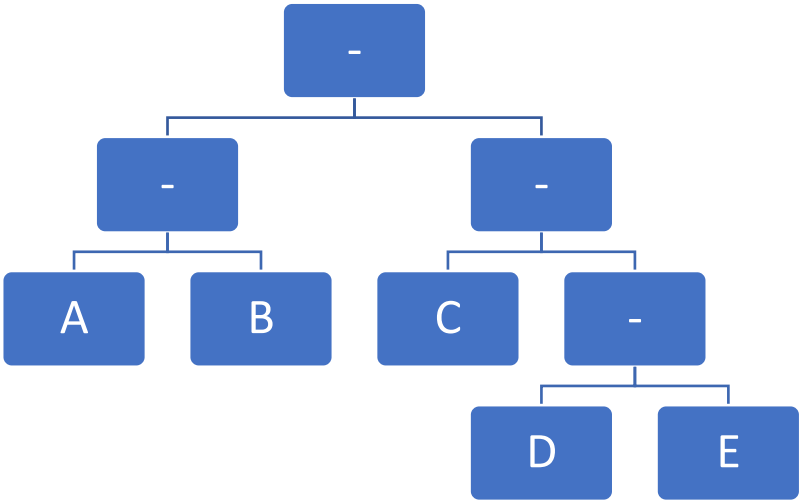
```
public class ArrayListStack<E> {
	private ArrayList<E> arraylist;

	public ArrayListStack(){ // COMPLETE }
	public void push(E info) { // COMPLETE }
	public E pop() {// COMPLETE }
}
```

NOTE: The class `ArrayList<E>` has the following methods, some of which can be useful in this problem:

- `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()`

Problem 3 (3 / 7 points)

The Huffman coding is a typical encoder to encode information. This coding generates a tree based on the probability of each symbol (e.g., letter) and the encoding can be obtained based on the tree. An example of tree is as follows:



In this tree, there a symbol (e.g., letter) in each leaf. In our implementation, we will represent them with `String`. The rest of the nodes are empty (in our case, we will put a dash, "-", in the information in those nodes). In order to find the encoding, we add a "0" each time we move left in the tree and we add a "1" each time we mode right in the tree. This way, the encoding of A is "00" (we move twice to the left) and the encoding of C is "10" (we move right first and then we move left).

Section 3.1 (0.5 points)

Indicate the encoding and the depth in the tree of the symbols B, D and E.

Section 3.2 (2 points)

Program the method `String findEncoding(String symbol)` in the class `LBTree` that represents the tree (you can find a partial implementation below). This method has a symbol as a parameter (e.g., "A") and returns a String with the corresponding encoding ("00" in the example"). If the symbol is not found in the tree (e.g., if "F" is used in the previous tree), the method will return -1.

Section 3.3 (0.5 points)

Implement the method `int findDepth(String symbol)`. This method indicates the depth of the node where a symbol is. If the symbol is not found, the method will return -1.

Hint: You can make use of the previous method.

```
public class LBTree<E> implements BTree<E> {
    private LBNode<E> root;

    public LBTree() {
        root = null;
    }

    public LBTree(E info) {
        root = new LBNode<E>(info, new LBTree<E>(), new LBTree<E>());
    }

    public boolean isEmpty() {...}
    public E getInfo() throws BTreeException {...}
    public BTree<E> getLeft() throws BTreeException {...}
    public BTree<E> getRight() throws BTreeException {...}
    public void insert(BTree<E> tree, int side) throws BTreeException {...}
    public BTree<E> extract(int side) throws BTreeException {...}
    public int size() {...}
    public String toString() {...}
    public int height() {...}
    public String findEncoding(String symbol) {// SECTION 3.2 }
    public int findDepth(String symbol) {// SECTION 3.3}
}
```

REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (1.5 points)

```
public Order search(int orderID) {
	Order result = null;
	if(!ordersToProcess.isEmpty()) {
		for(int i=0; i<ordersToProcess.size(); i++) {
			Order aux = ordersToProcess.dequeue();
			if(aux.getOrderID() == orderID) {
				result = aux;
			}
			ordersToProcess.enqueue(aux);
		}
	}
	return result;
}
```

Evaluation criteria:

- 0.2: Create variable and return it at the end
- 0.2: Loop to iterate over the size of the queue
- 0.3: Dequeue element
- 0.3: If to check if the orderIDs match.
- 0.2: Assign the order to the result variable in case the orderID matches
- 0.3: Enqueue element. If there is a break to stop enqueuing after finding the element, max 0.1
- Significant errors are subject to additional penalties

Section 1.2 (1 point)

```
public Order search(int orderID) {

	Node<Order> aux = head;
	while(aux != null) {
		if(aux.getInfo().getOrderID()==orderID) {
			return aux.getInfo();
		}
		aux = aux.getNext();
	}
	return null;
}
```

Evaluation criteria

- 0.2: Initialize aux node
- 0.3: Loop to traverse the list (including update of aux)
- 0.3: Check if the ID matches
- 0.2: Return the information correctly
- Significant errors are subject to additional penalties

Section 1.3 (0.5 points)

```
public Order search2(int orderID) {
	return ordersToProcess.search(orderID);
}
```

Evaluation criteria

- 0.5: Correct call to method search. If return is not used, máx 0.2.
- Significant errors are subject to additional penalties

PROBLEM 2

```
public class ArrayListStack<E> {
	private ArrayList<E> arraylist;

	public ArrayListStack(){
		arraylist = new ArrayList<E>();
	}

	public void push(E info) {
		arraylist.add(0, info); // or arraylist.add(info)
	}

	public E pop() {
		if(!arraylist.isEmpty())
		   return arraylist.remove(0);//or arraylist.remove(arraylist.size()-1)
		else
		   return null;
	}}
```

Evaluation criteria

- 0.2: Constructor
- 0.4: Method push
- 0.4: Method pop (penalize 0.1 if the case when the list is empty is not checked)
- Significant errors are subject to additional penalties

PROBLEM 3

Section 3.1 (0.5 points)

Symbol	Encoding	Depth
B	01	2
D	110	3
E	111	3

Evaluation criteria

- Penalize 0.1 for each incorrect cell (e.g., 0.4 if 5/6 cells are correct and 0 is less 2 cells are correct)
- Significant errors are subject to additional penalties

Section 3.2 (2 points)

```
public String findEncoding(String symbol) {
	if(isEmpty()) { // Base case when reaching a leaf without finding the symbol
		return "-1";
	} else if(root.getInfo().equals(symbol)) { // Base case when finding the
symbol
		return "";
	} else {
		// Add 0 when moving to the left
		String left = "0" + root.getLeft().findEncoding(symbol);
		// Add 1 when moving to the right
		String right = "1" + root.getRight().findEncoding(symbol);

		// If the left String does not contains -1,
		// it means the symbol is found in the left part and it is returned
		if(!left.contains("-1")) return left;
		// If the right String does not contains -1,
		// it means the symbol is found in the left part and it is returned
		if(!right.contains("-1")) return right;
		// If both left and right contain -1, it means the symbol is not found
and -1 is returned
		return "-1";
	}
}
```

Evaluation criteria

- 0.3: Base case when reaching a leaf without finding the symbol
- 0.3: Base case when finding the symbol
- 0.4: Recursive case when moving to the left
- 0.4: Recursive case when moving to the right
- 0.6: Correct management to return the result depending on the part of the tree where symbol is found
- Significant errors are subject to additional penalties

Section 3.3 (0.5 points)

```
public int findDepth(String symbol) {
	String coding = findEncoding(symbol);
	if(coding.equals("-1")) return -1;
	return coding.length();
}
```

Evaluation criteria

- 0.2: Call to findEncoding
- 0.1: Check if coding is -1, and return -1 in that case
- 0.2: Return the information when coding is not -1
- Significant errors are subject to additional penalties