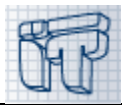


FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second midterm exam

Second Part: Problems (7 points out of 10)

Duration: 90 minutes
Highest score possible: 7 points
Date: May 4, 2018

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 want to implement a basic queue (`Queue<E>`) using a circular array (`ArrayQueue<E>`) whose capacity will be determined at creation time. Being a queue, elements will be inserted by one end (referenced by the attribute `tail`) and extracted by the other end (referenced by the attribute `head`). As it is implemented through a circular array, if `tail` reaches the last position of the array, and there are free spots at the beginning, the following element will be inserted in the first position of the array (updating, of course, the corresponding value of `tail`). Under no circumstances may the number of elements in the array exceed its capacity. The following table presents the interface `Queue<E>` and the structure of the class `ArrayQueue<E>`.

IMPORTANT NOTE: The class `ArrayQueue<E>` includes all the attributes and methods you must use/implement to solve this problem. You are not allowed to create additional attributes or methods.

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

```
public class ArrayQueue<E> implements Queue<E> {
    private E[] data;
    private int head = -1;
    private int tail = -1;
    public ArrayQueue(int capacity) {
        this.data = (E[]) new Object[capacity];
    }
    public int size() { // Section 1.1 }
    public void enqueue(E info) { // Section 1.2 }
    public E dequeue() { // Section 1.3 }
}
```

Section 1.1 (0.5 points)

Implement the method `size()`, which calculates the size of the queue. Be aware of the different values `head` and `tail` may take.

Section 1.2 (1.25 points)

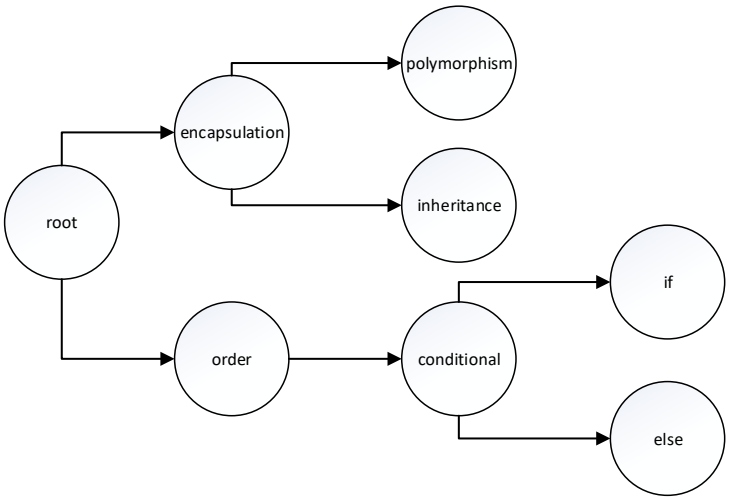
Implement the method `enqueue(E info)`, which inserts a new element in the queue. In case the queue is full at the time of inserting the new element print an error message on screen.

Section 1.3 (1.25 points)

Implement the method `dequeue()`, which extracts an element from the queue, returning its information and setting the corresponding position to `null`. In case the queue is empty, return `null`.

Problem 2 (3 / 7 points)

In a MOOC (Massive Open Online Course), social interactions mostly occur in the course forum. In this problem, we want to focus on identifying the skills that appear more in forum threads, because that could serve to detect if learners have difficulties in grasping some concepts. The skills of the course can be modelled in a tree, so that each skill can have several subskills. For example, "inheritance" and "polymorphism" can be related to "encapsulation". An example of a skills tree is as follows:



To model these skills, the class `Skill` has been implemented. This class has four attributes: (1) `name`, (2) `mentions`, (3) `subskills` and (4) `color`.

`name` represents the skill (e.g., "polymorphism"), `mentions` indicates the number of messages that contains the skill (you do not need to care about how that value is obtained), and `subskills` is an `ArrayList`, which contains the subskills of a certain skill. Apart from that, `color` indicates a RGB value that serves to represent the number of times a skill appears. Skills that appear many times will be represented with reddish colors (they can generate many doubts) and skills that rarely appear will be represented with greenish colors. However, colors are not defined initially and they need to be calculated.

Moreover, there is a class `SkillsTree` to model the tree, which only contains the reference to the root, which is a default node that is used to build the tree upon it. The tree structure defined in this class is useful to color the skills according to the following criteria:

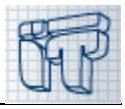
1) Colors are first assigned to **leaf** nodes. A method has already been implemented to compute RGB values (`computeRGB(int mentions, int max, int min)`) . This method requires the number of mentions of a given leaf skill (`mentions`) and the number of mentions of the skill with most/least mentions (**among leaf skills**), which are `max` and `min`, respectively.
2) Parent nodes should be colored by taking the average color of their children. This means that if we have a node with two children whose colors are (4,2,0) and (8,0,0), the color will be (6,1,0). You need to round values down when you compute the average.

Section 2.1 (1 point)

Implement the method `insert(Skill s, String parent)`, which inserts a skill in the `ArrayList` of subskills of another skill, whose name is given. If the parent is not found, the method does not do anything. You must use recursion in the implementation of the method.

Section 2.2 (2 points)

Implement the method `addColors(Skill root_skill, int max, int min)`, which sets the attribute of color of each skill according to the abovementioned criteria. This method is called from another method `addColors()`, which calls this method using the `root` skill and computes the `max` and `min`



variables of the tree. You can also assume that there are always two skills at least. You must use recursion in the implementation of the method.

You are given the structure of the class `Skill` and `SkillsTree` as a reference.

```
public class Skill {
 private String name;
 private int mentions;
 private int[] color;
 private ArrayList<Skill> subskills;

 public Skill(String name, int mentions){
    this.name = name;
    this.mentions = mentions;
    this.color = new int[3];
    this.subskills = new ArrayList<Skill>();
 }

 public String getName() {return name;}
 public int getMentions() {return mentions;}
 public int[] getColor() {return color;}
 public void setColor(int[] color) {
    this.color = color;}
 public ArrayList<Skill> getSubskills() {
    return subskills;
 }
 public void addSubskill(Skill s){
    subskills.add(s);
 }
}
```

```
public class SkillsTree {
 private Skill root;

 public SkillsTree() {
     root = new Skill("root", 0);
 }

 public void insert(Skill s, String
parent){
   // SECTION 2.1
 }

 public int getMinValue(){...}
 public int getMaxValue(){...}

 public void addColors(){
  int max = getMaxValue();
  int min = getMinValue();
  addColors(root, max, min);
 }

 public int[] addColors(Skill
root_skill, int max, int min){
   // SECTION 2.2
}
```

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

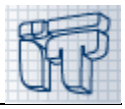
- `boolean add(E e)`
- `void add(int index, E element)`
- `E get(int index)`
- `int indexOf(Object o)`
- `boolean isEmpty()`
- `E set(int index, E element)`
- `int size()`

Problem 3 (1 / 7 points)

Implement the method `void selectionSort (String[] s)`, which sorts an array of `String` using the Selection Sort algorithm. This method should use the version of the Selection Sort algorithm which sorts in descending order (from highest to lowest), searching for the highest String in the unsorted part of the array and swapping it with the first unsorted element.

Note 1: You can call the method `void swap(String[] s, int i, int j)` as part of your method, which swaps elements in positions `i` and `j` of the array `s`. Assume this method is already implemented, so you do not need to code it.

Note 2: Remember that Strings shall be compared with the method `compareTo(String anotherString)`



REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (0.5 points)

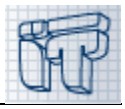
```
public int size() {
    if (head==-1){
        return 0;
    }
    if (tail >= head){
        return (tail-head +1);
    } else{
        return (tail-head + 1 + data.length);
    }
}
```

- 0.1: case 1, empty queue
- 0.2: case 2, tail greater than head
- 0.2: case 3, head greater than tail
- The case tail == head happens when there is only one element in the queue, and can be part of case 2 or 3.

```
Alternative (less efficient)
public int size() {
    int count = 0;
    for (int i = 0; i < data.length -1; i++){
        if (data[i] != null){
            count++;
        }
    }
    return count
}
```

Section 1.2 (1.25 points)

```
public void enqueue(E info) {
    if (head == -1){
        data[0] = info;
        head = 0;
        tail = 0;
    } else {
        if (tail + 1 == head){
            System.err.println("Stack Overflow");
        } else if (head == 0 && tail == data.length - 1){
             System.err.println("Stack Overflow");
        } else{
            if (tail == data.length - 1){
                tail = 0;
            } else {
                tail ++;
            }
            data[tail] = info;
        }
    }
}
```



- 0.25: case 1, empty queue
- Máx 1: case 2, non-empty queue
    - 0.25 if considering the two cases where exceeding the capacity of the queue (0.15 if considering only one case)
    - 0.75 if considering the case where not-exceeding the capacity of the queue
        - 0.5 updating the value of tail
            - 0.25 updating the value of tail when tail is less than data.length-1
            - 0.25 updating the value of tail when tail is data.length-1
        - 0.25 storing the information correctly

Section 1.3 (1.25 points)

```
public E dequeue() {
    E info;
    if (head == -1){
        info = null;
    } else if (head == tail){
        info = data[head];
        data[head]=null;
        head = -1;
        tail = -1;
    } else {
        info = data[head];
        data[head]=null;
        if (head == data.length-1){
            head = 0;
        } else {
            head++;
        }
    }
    return info;
}
```

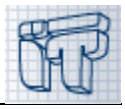
- 0.25: case 1, empty queue
- Máx 1: case 2, non-empty queue
    - 0.25 if considering the case where there is only one element in the queue
    - 0.75 if considering the case where there is more than one element in the queue
        - 0.5 updating the value of head
            - 0.25 updating the value of head when head is less than data.length-1
            - 0.25 updating the value of head when head is data.length-1
        - 0.25 returning the information correctly

PROBLEM 2

Section 2.1 (1 point)

```
public void insert(Skill s, String parent){
      insert(s, root, parent);
}

public void insert(Skill s, Skill root_skill, String parent){
      if(root_skill.getName().equals(parent)){
            root_skill.addSubskill(s);
      } else {
```



```
            for(int i=0; i<root_skill.getSubskills().size();i++){
                  insert(s, root_skill.getSubskills().get(i), parent);
            }
      }
}
```

Section 2.1 (1 points):

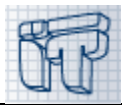
- 0.10: Call to the auxiliary method
- 0.05: Auxiliary method signature
- 0.35: Base case
  - 0.20: Check when the skill name and the parent match
  - 0.15: Add subskill
- 0.50: Recursive case
  - 0.15: For to iterate over children
  - 0.35: Recursive call
- Significant errors are subject to additional penalties

Section 2.2 (2 points)

```
public int[] addColors(Skill root_skill, int max, int min){
      if(root_skill.getSubskills().size() > 0){
            int r = 0;
            int g = 0;
            int b = 0;
            for(int i=0; i<root_skill.getSubskills().size();i++){
                  Skill new_root = root_skill.getSubskills().get(i);
                  int[] colors_child = addColors(new_root, max, min);
                  r += colors_child[0];
                  g += colors_child[1];
                  b += colors_child[2];
            }
            r = r/root_skill.getSubskills().size();
            g = g/root_skill.getSubskills().size();
            b = b/root_skill.getSubskills().size();
            root_skill.setColor(new int[]{r,g,b});
            return root_skill.getColor();
      } else {
            int[] rgb = computeRGB(root_skill.getMentions(), max, min);
            root_skill.setColor(rgb);
            return root_skill.getColor();
      }
}
```

Section 2.2 (2 points)

- 0.6: Base case
  - 0.2: Identification of the base case when a leaf node is reached
  - 0.1: Call to the computeRGB method
  - 0.1: Set the color of the skill
  - 0.2: Return color
- 1.4: Recursive case
  - 0.3: For to iterate over all children
  - 0.5: Get colors from child in and store it in a variable. If return is used at this point and therefore only the for only applies to the first child, max 0.15.
  - 0.3: Compute the average of RGB
  - 0.1: Set the color of the skill
  - 0.2: Return color



- Significant errors are subject to additional penalties

PROBLEM 3

```
void selectionSort (String[] s) {
    for (int i=0; i<s.length; i++) {
        int m = i;
        for (int j=i; j<s.length; j++) {
            if (s[j].compareTo(s[m])>0){
                m = j;
            }
        }
        swap(s, i, m);
     }
}
```

- 0.25: External loop, from 0 to s.length or s.length-1
- 0.25: Internal loop from i to s.length
- 0.25: Comparison of strings and getting the maximum into a local variable
- 0.25 Calling the method that swaps correctly