Bachelor in Informatics Engineering	Algorithms and Data Structures **19th of May, 2016.** **FINAL EXAM, ORDINARY CALL**

Name: ....

Surname: ....

PLEASE, READ CAREFULLY THE FOLLOWING INSTRUCTIONS BEFORE START

1. All student data asked in this document must be filled. Please, use a pen to write in this document (not a pencil)
2. This document and all additional sheets must be delivered upon completion.
3. Please, add page numbers in all pages before delivering.
4. You cannot leave the classroom until the exam has ended.
5. Mobile phones must be disconnected during the exam.
6. The exam will take **3 hours**.
7. **In this exam, the use of classes provided by the Java API (ArrayList, LinkedList, etc.). The use of Arrays is also forbidden.**

INTRODUCTION:

"Metro de Madrid" is a metropolitan railway suffering continuous changes: new stations are built, others are closed (temporarily or permanently). Currently there are 301 metro stations, but this number varies very frequently.

PROBLEM 1 (1 point)
To store alphabetically the list of metro stations you can use an array (static structure) or a linked list (dynamic structure). Answer the following questions:

a) What is the complexity of adding a new station in an array? And for a linked list? (0.3 points).
Solution:
Considering an array, the worst case would be when there are no more memory positions to store the new station in the array. In this case, it would be needed to allocate memory space to store the extended array and, then, move all the elements from the older array to the new one, assuring the alphabetical order.
In a singly linked list, the worst case would be when the new station has to be added at the end of the list so the whole list must be traversed. In this situation complexity would be linear, O(n).

b) If we know the position of the station into an array or a list (that is, the index for that station) what is the complexity of querying a given station in an array? And, for a linked list? (0.3 points).
Solution:
Considering an array, complexity to query an element in a given position is constant O(1) because it is possible accessing directly an element taking into account the memory position (index) of the element.
In a singly linked list, all nodes must be traversed until the one to be returned is found. In the worst case this will be the last node. So complexity is linear O(n).

c) Which structure would you recommend if most of the operations to be executed on that structure are going to be read operations? And if removal and update operations are considered? (0.4 points).

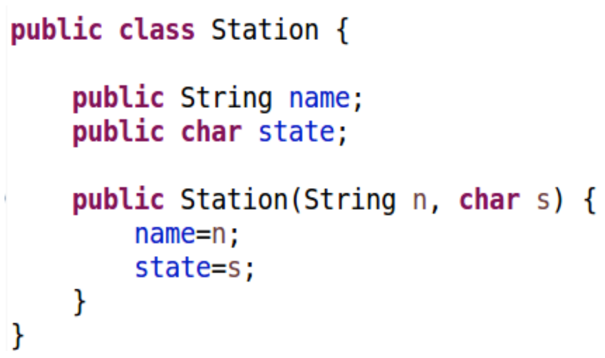
Solution:

If most operations are read operations the most adequate structure is an array, because it possible to have a direct access to the element.

If most operations are insertions and deletions, a singly linked list is more efficient because of the dynamic management of memory space.

PROBLEM (2 points)

To make the problem easier, lets suppose that for each station only the name and state (open or temporarily closed) are needed. Please, find attached an implementation of the Station class though you can modify it to include new attributes or methods, if needed, in order to solve the problem:



```
public class Station {

    public String name;
    public char state;

    public Station(String n, char s) {
        name=n;
        state=s;
    }
}
```

Create a class Metro implementing a singly linked list to store the list of stations. You do not have to implement all the methods for a singly linked list, only the following one is needed:

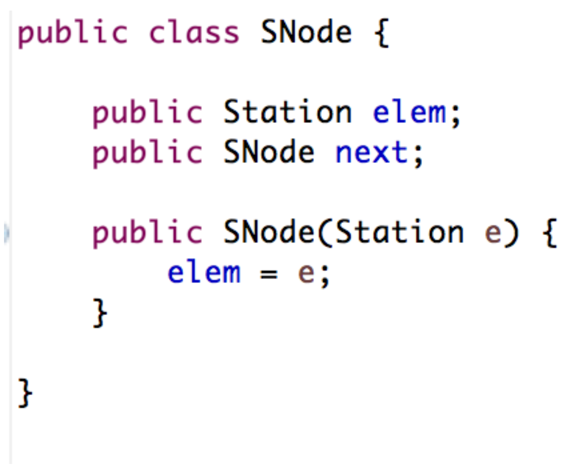
- **boolean changeStatus(String nameStation, char state)**: the final goal of this method is to change the state of a given station or even to remove the given station if the permanent closing of the station is ordered. The method receives the name of the station to be searched on the list and an state argument of type 'char' saying the type of operation to be performed:
    - If state = 'o', the method must search for the station and change the state to open ('o')
    - If state = 'c', the method must search for the station and change the state to closed ('c')
    - If state = 'b', the state of the station must not be changed but it must be removed from the list of stations.

  The method returns true if the operation has been properly executed and false if the station is already in the new state or if it is not found in the list of stations.

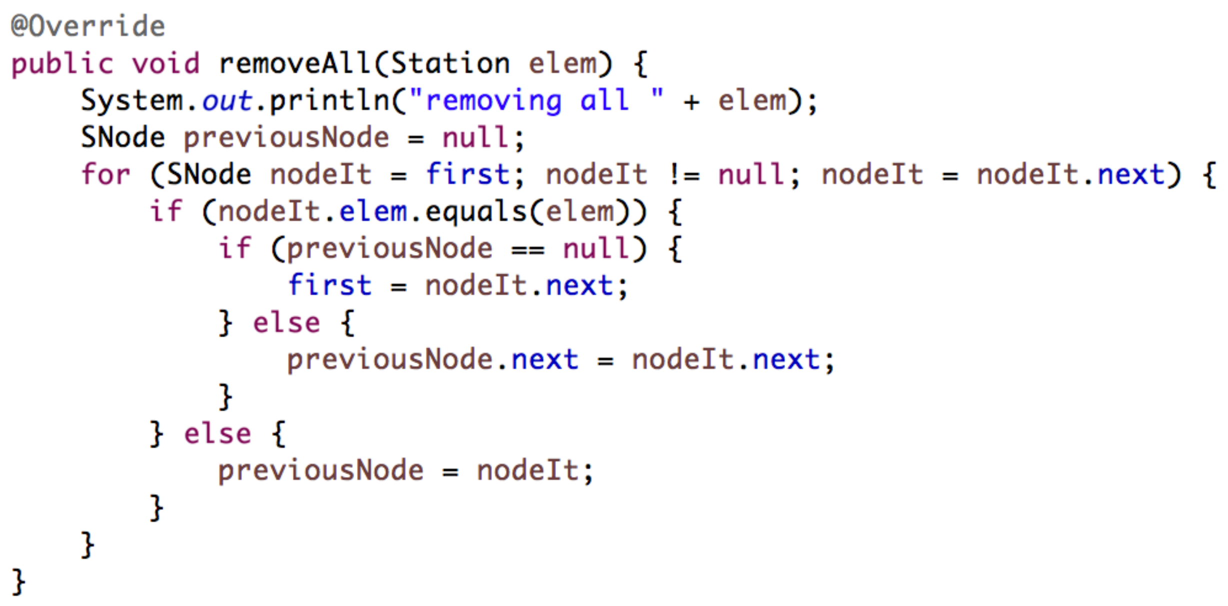
Note 1: If your solution requires other methods in the IList interface (for example, isEmpty(), getSize(), getIndexOf(), removeAt(), etc), **you must include their implementation**.

Note 2: Solutions based on arrays will not be evaluated.

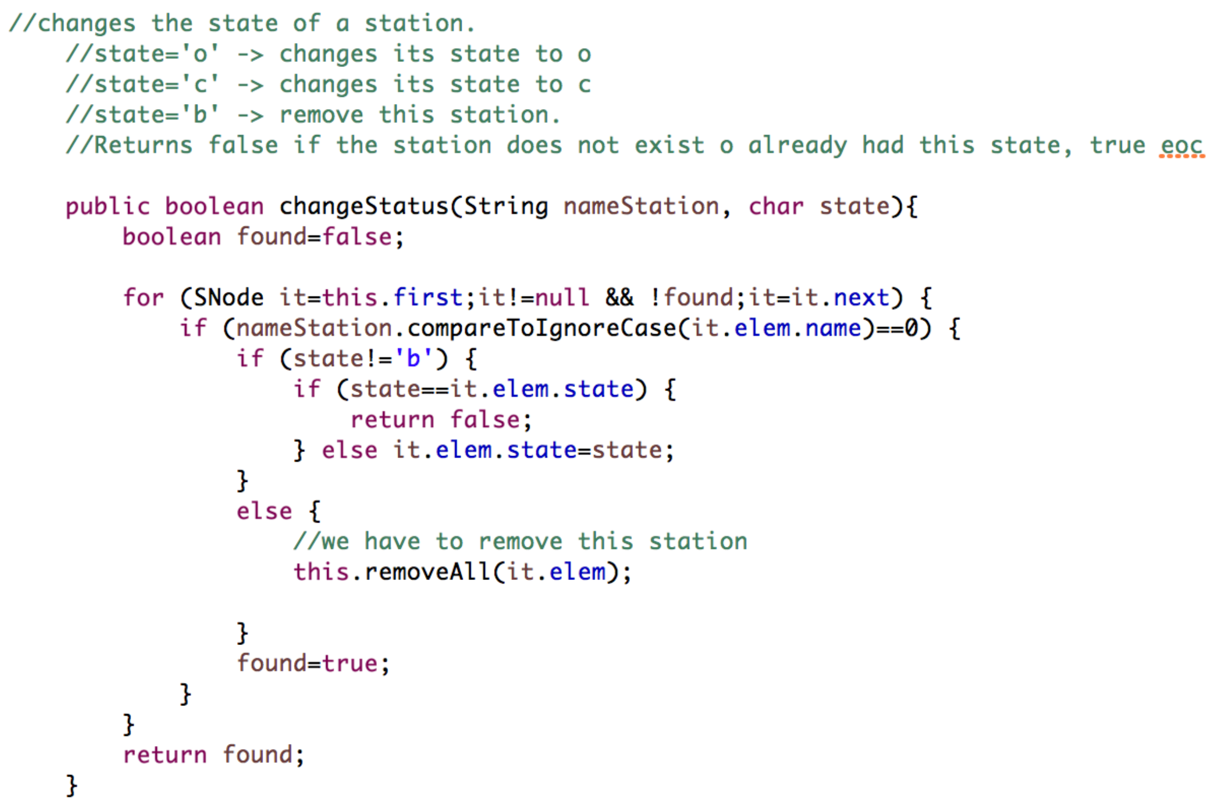
Solution:



```java
public class SNode {

    public Station elem;
    public SNode next;

    public SNode(Station e) {
        elem = e;
    }

}
```



```java
@Override
public void removeAll(Station elem) {
    System.out.println("removing all " + elem);
    SNode previousNode = null;
    for (SNode nodeIt = first; nodeIt != null; nodeIt = nodeIt.next) {
        if (nodeIt.elem.equals(elem)) {
            if (previousNode == null) {
                first = nodeIt.next;
            } else {
                previousNode.next = nodeIt.next;
            }
        } else {
            previousNode = nodeIt;
        }
    }
}
```



```java
//changes the state of a station.
    //state='o' -> changes its state to o
    //state='c' -> changes its state to c
    //state='b' -> remove this station.
    //Returns false if the station does not exist o already had this state, true eoc

    public boolean changeStatus(String nameStation, char state){
        boolean found=false;

        for (SNode it=this.first;it!=null && !found;it=it.next) {
            if (nameStation.compareToIgnoreCase(it.elem.name)==0) {
                if (state!='b') {
                    if (state==it.elem.state) {
                        return false;
                    } else it.elem.state=state;
                }
                else {
                    //we have to remove this station
                    this.removeAll(it.elem);

                }
                found=true;
            }
        }
        return found;
    }
```

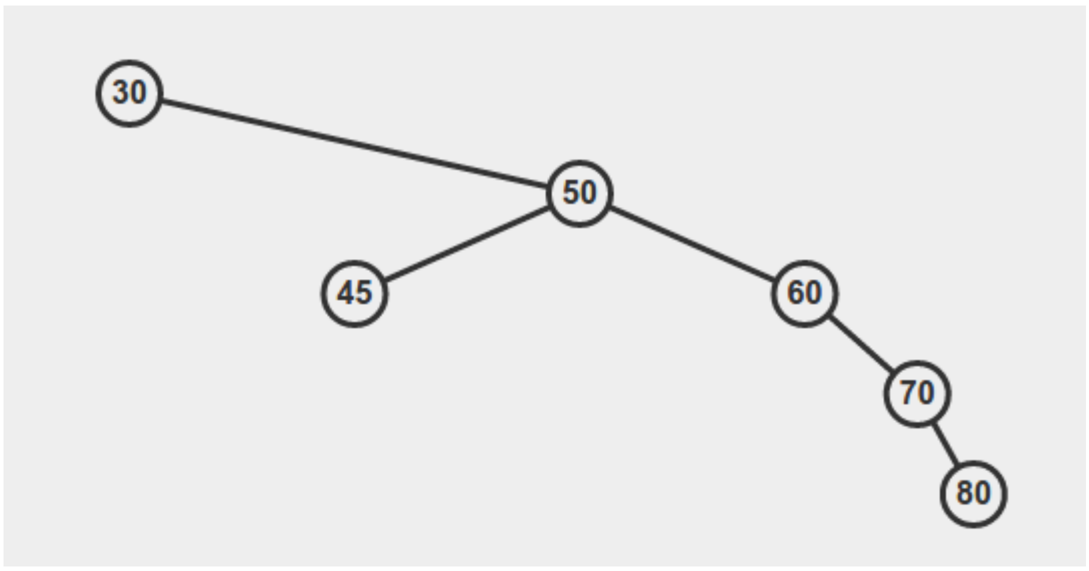
PROBLEM 3 (2 points). Implement a method receiving an String array and returning an alphabetically ordered array. The method must implement the **quicksort** algorithm.

Solution:

```java
public class QuickSort {


	public static void print(String vector[]){
		for(int i=0;i<vector.length;i++)
			System.out.print(vector[i]+"-");
		System.out.println();
	}

	public static void quicksort(String vector[], int left, int right) {

		  String pivot=vector[left]; //first element is the pivot (maybe not the best strate-
gy)
		  int i=left; // i performs the search from left to right
		  int j=right; // j performs the search from right to left
		  String aux;
		  System.out.println("------------------------------------------------------");
		  System.out.print("QUICKSORT RECURSIVE CALL. ");

		  System.out.println("SUB-ARRAY " + "["+ vector[i] + " hasta "+ vector[j]+ "]");
		  System.out.println("PIVOT IS " + pivot);


		  while(i<j){            // while searches were not crossed

			while(vector[i].compareToIgnoreCase(pivot) <=0 && i<j) i++; // the greater
element than the pivot is searched

			while(vector[j].compareToIgnoreCase(pivot) > 0) j--;         // the lower
element than the pivot is searched
		     if (i<j) {                           // if no crossed
			 aux= vector[i];                        // are swapped
				System.out.print("LOWER AND GREATER THAN PIVOT VALUES ARE CHANGED:
");

				System.out.println(vector[j] + " Y " + aux);


			 vector[i]=vector[j];
			 vector[j]=aux;
		     }
		   }


		System.out.println("THE PIVOT IS CHANGED WITH THE POSITION WHERE THE LOOP STOPED: "
+  pivot + " CHANGES WITH " + vector[j]);

		  vector[left]=vector[j]; // the pivot is put in the corresponding place
		  vector[j]=pivot; // lower elements to the left and higher elements to the right

		  System.out.print("ARRAY IS: ");
		  print(vector);

		   if(left<j-1)
		      quicksort(vector,left,j-1); // left subarray is sorted
		   if(j+1 <right)
		      quicksort(vector,j+1,right); // right subarray is sorted
		}
		public static void main(String[]args){

			String[] a={"C","A","X","F","G"};

			System.out.print("EL ARRAY ES: ");
			  print(a);
```

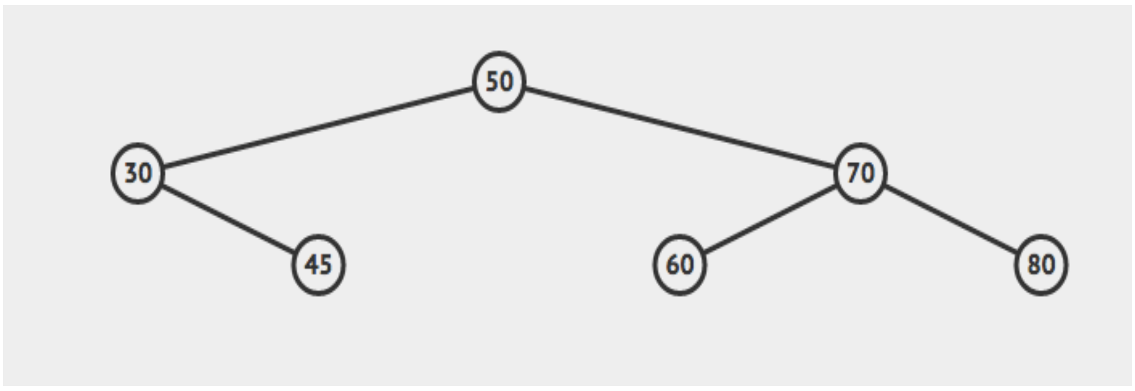
```
                quicksort(a,0,a.length-1);
                print(a);
        }
}
```

PROBLEM 4 (1 point). Given the following Binary Search Tree (BST):

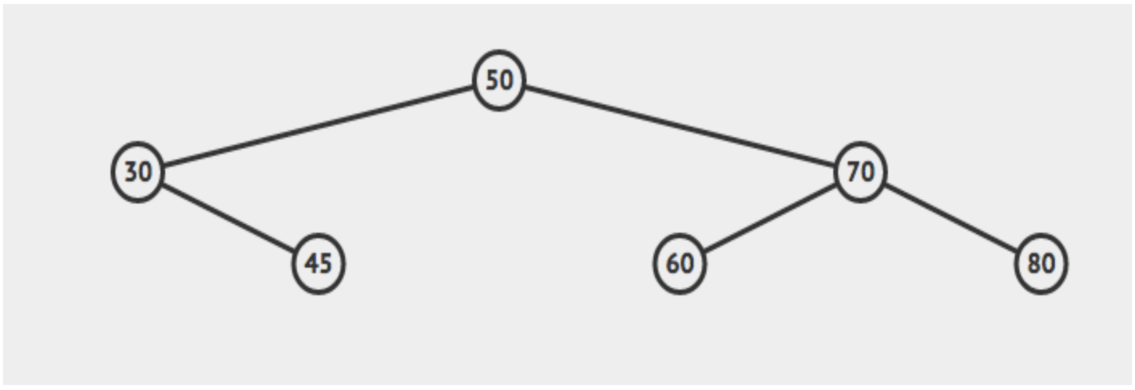


a) (0.4 points) Draw the BST obtained after applying the necessary steps to obtain the equivalent perfectly balanced tree (Size Balanced Tree).

Solution:



b) (0.4 points) Draw the BST obtained after applying the necessary steps to obtain the equivalent AVL tree (Height Balanced Tree).
Solution:



c) (0.2 points) What is the main goal pursued by balancing operations in BST?

Solution:
Having a balanced tree (in size or height) makes that operations on the tree remain with logarithmic complexity.

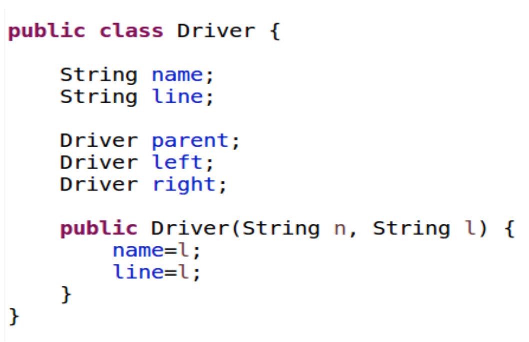
PROBLEM 5 (2.5 points)

A key element for the proper operation of “Metro de Madrid” is the correct management of the staff of drivers. For each driver is necessary to know its name and the line to which it is assigned.

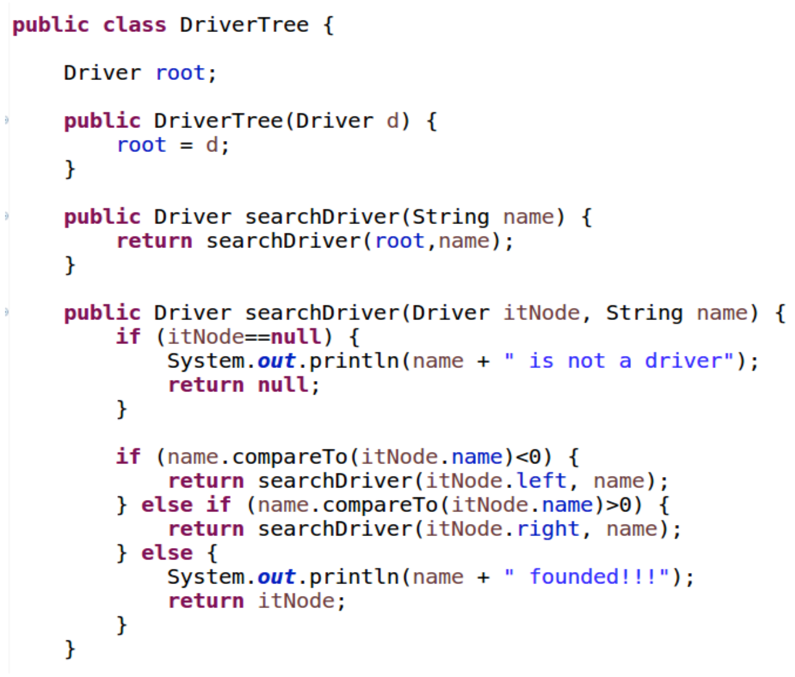
a) (0.5 points) What are the advantages of storing the set of drivers in a Binary Search Tree (BST) against a Linked List structure?

Solution:
In a binary search tree complexity for removal, insertion and search operations is logarithmic while linked list have linear complexity.

The following images show Driver class and DriverTree class, that allow to represent a driver and a Binary Search Tree to store the set of all drivers, respectively. The DriverTree class is incomplete because it only includes the constructor method and a method for searching for a specific driver by its name.



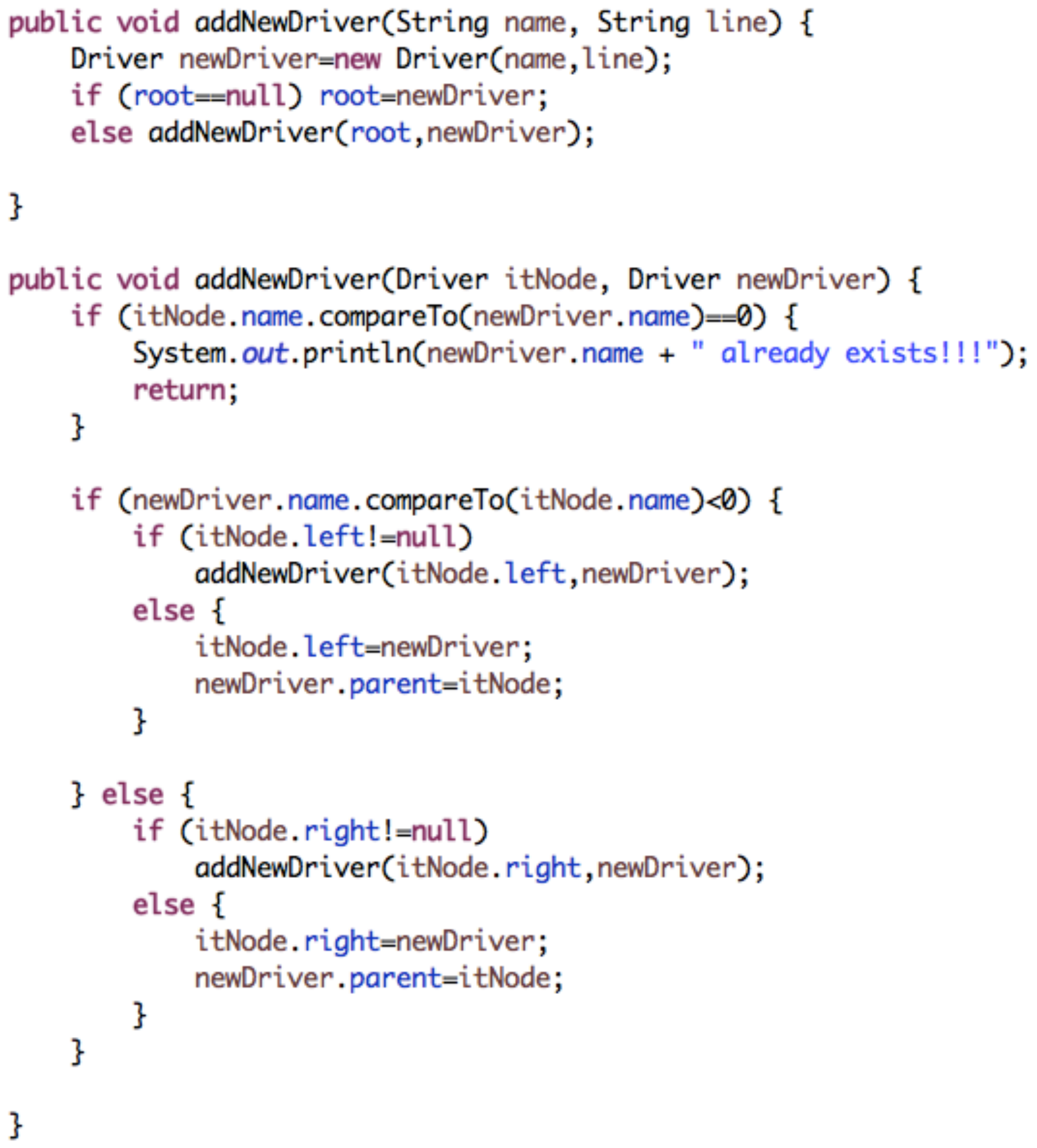
```java
public class Driver {

    String name;
    String line;

    Driver parent;
    Driver left;
    Driver right;

    public Driver(String n, String l) {
        name=l;
        line=l;
    }
}
```



```java
public class DriverTree {

    Driver root;

    public DriverTree(Driver d) {
        root = d;
    }

    public Driver searchDriver(String name) {
        return searchDriver(root,name);
    }

    public Driver searchDriver(Driver itNode, String name) {
        if (itNode==null) {
            System.out.println(name + " is not a driver");
            return null;
        }

        if (name.compareTo(itNode.name)<0) {
            return searchDriver(itNode.left, name);
        } else if (name.compareTo(itNode.name)>0) {
            return searchDriver(itNode.right, name);
        } else {
            System.out.println(name + " founded!!!");
            return itNode;
        }
    }
```

b) (2 points) Implement **void addNewDriver(String name, String line)** method to add a new driver to the Binary Search Tree that stores the drivers.

Solution:

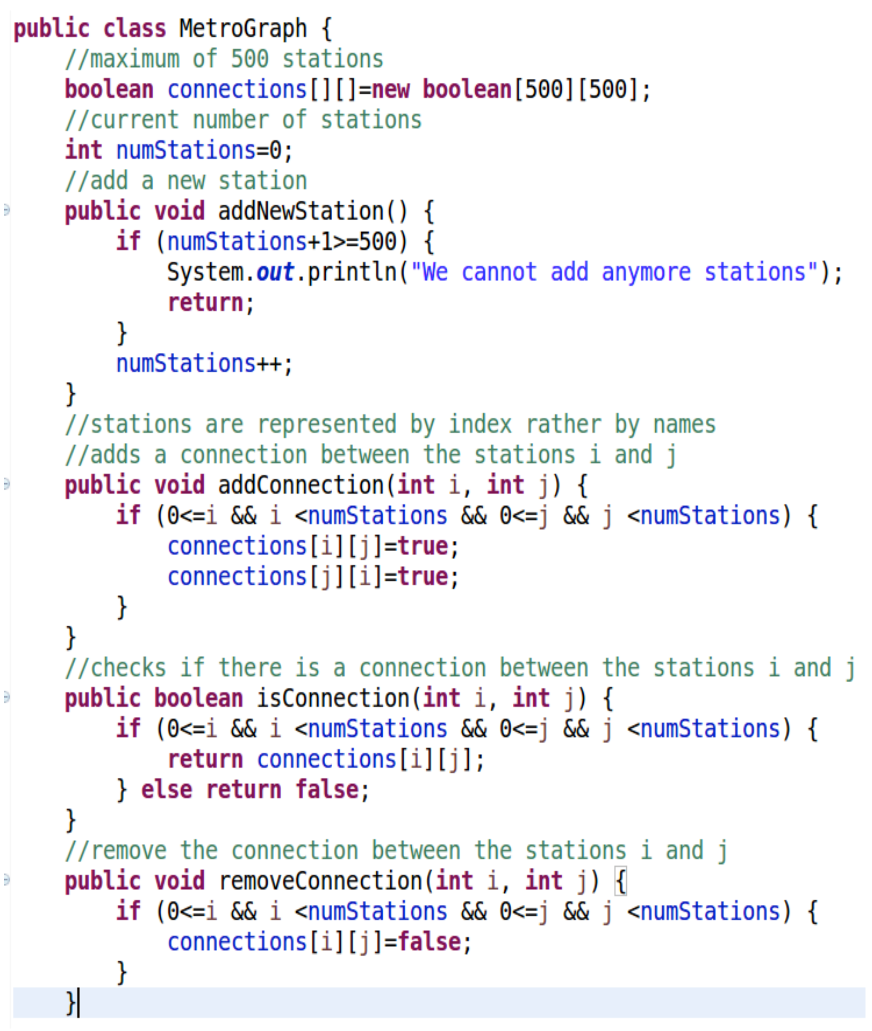


```java
public void addNewDriver(String name, String line) {
    Driver newDriver=new Driver(name,line);
    if (root==null) root=newDriver;
    else addNewDriver(root,newDriver);

}

public void addNewDriver(Driver itNode, Driver newDriver) {
    if (itNode.name.compareTo(newDriver.name)==0) {
        System.out.println(newDriver.name + " already exists!!!");
        return;
    }

    if (newDriver.name.compareTo(itNode.name)<0) {
        if (itNode.left!=null)
            addNewDriver(itNode.left,newDriver);
        else {
            itNode.left=newDriver;
            newDriver.parent=itNode;
        }

    } else {
        if (itNode.right!=null)
            addNewDriver(itNode.right,newDriver);
        else {
            itNode.right=newDriver;
            newDriver.parent=itNode;
        }
    }

}
```

PROBLEM 6 (1.5 points).

Metro network can be implemented with a graph, where the stations are the graph nodes (Vertices) and the edges (Arcs) would be connections between stations. To simplify the problem, stations instead of being represented by their name, are identified with an index.

The following image shows the (incomplete) implementation of a network of stations based on an adjacency matrix. This implementation contains some methods as **addNewStation** (which adds a new station), **AddConnection** (which adds a connection between a pair of stations), **isConnection** (method that checks if there is a direct connection between two stations) and **removeConnection** (method that eliminates the connection between two stations).

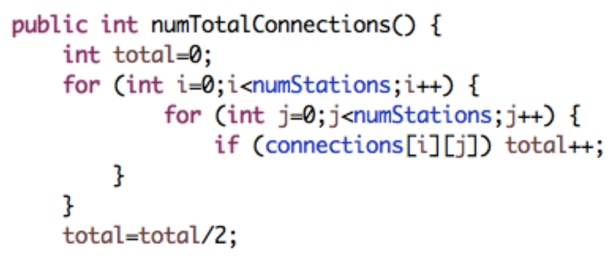


```java
public class MetroGraph {
    //maximum of 500 stations
    boolean connections[][]=new boolean[500][500];
    //current number of stations
    int numStations=0;
    //add a new station
    public void addNewStation() {
        if (numStations+1>=500) {
            System.out.println("We cannot add anymore stations");
            return;
        }
        numStations++;
    }
    //stations are represented by index rather by names
    //adds a connection between the stations i and j
    public void addConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            connections[i][j]=true;
            connections[j][i]=true;
        }
    }
    //checks if there is a connection between the stations i and j
    public boolean isConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            return connections[i][j];
        } else return false;
    }
    //remove the connection between the stations i and j
    public void removeConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            connections[i][j]=false;
        }
    }
```

a) (0.75 points) Implement a method to calculate the total number of connections in the Metro network.

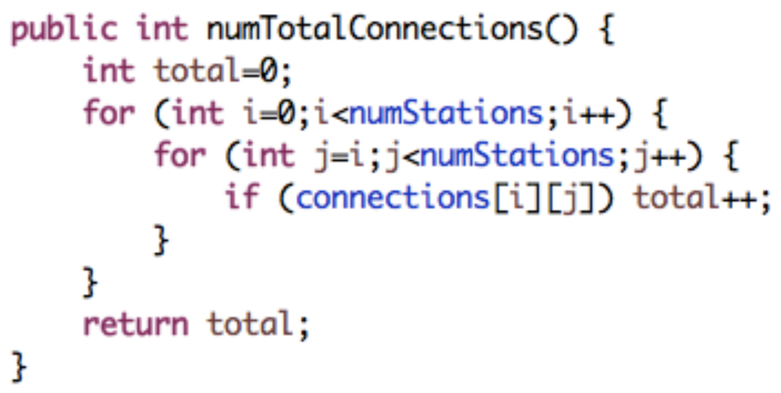
Solution:

A metro should be represented as a non directed graph because stations are connected in both directions. In this case the solution would be:



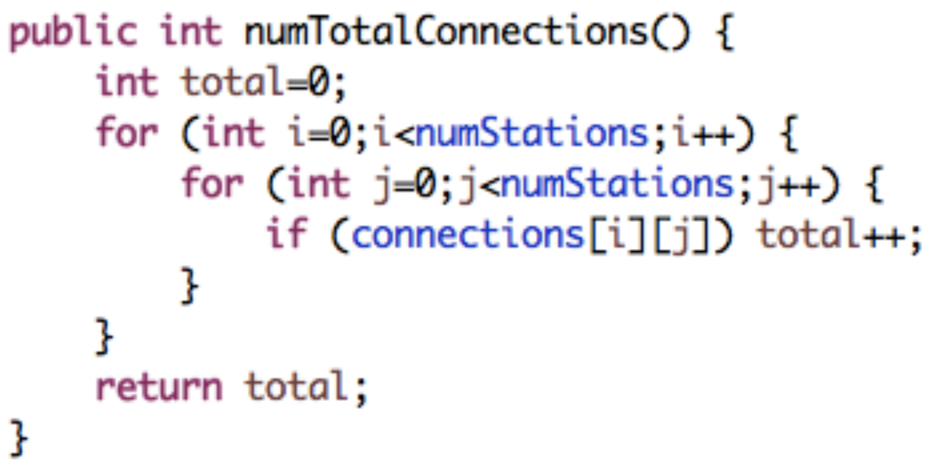
```java
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
            for (int j=0;j<numStations;j++) {
                if (connections[i][j]) total++;
        }
    }
    total=total/2;
```

A more efficient solution would be:



```java
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
        for (int j=i;j<numStations;j++) {
            if (connections[i][j]) total++;
        }
    }
    return total;
}
```

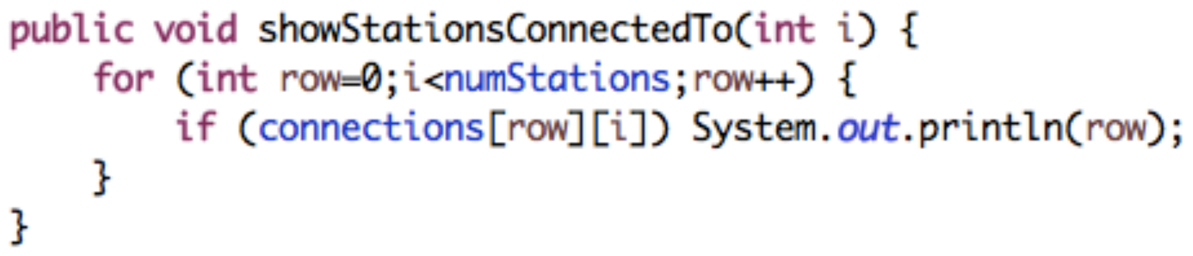
Because in a non directed graph the adjacency matrix is symmetric.
On the other hand, the provided remove method could lead us think that the graph is directed because only one direction is removed. This representation makes sense if a connection in a given direction between two stations can be closed. In this case, the solution would be:



```java
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
        for (int j=0;j<numStations;j++) {
            if (connections[i][j]) total++;
        }
    }
    return total;
}
```

b) (0.5 points) Implement a method that receives an index (represents a station) and display on the screen the indexes of the stations that have direct connection with it.

Solution:



```java
public void showStationsConnectedTo(int i) {
    for (int row=0;i<numStations;row++) {
        if (connections[row][i]) System.out.println(row);
    }
}
```

c) (0.25 points) During the course we studied two types of implementations for graphs: based on **adjacency matrix** and based on **adjacency lists**. What are the advantages of graph's implementation based on adjacency list versus graph's implementation based on adjacency matrix?

Solution:
An implementation based in adjacency lists requires less memory space because only information about existing edges is stored. Besides, some operations (for example, traversing the edges) are computationally harder for adjacency matrix than those implemented through an adjacency list.