 **Bachelor in Informatics Engineering**	Algorithms and Data Structures **27th of May, 2015.** **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.
2. The exam includes 3 problems. The final mark will be over 6 points. A 40% of the mark is needed to get benefit from continuous assessment (so, at least 2,4 points over 6 are needed).
3. Those students that do not want following continuous assessment still hold the right to make a final exam (ordinary exam + additional exercises). These exercises will be delivered after this first phase (and additional time will be scheduled).
4. Only the answer written in this document will be evaluated.
5. This document and all additional sheets must be delivered upon completion.
6. You cannot leave the classroom until the exam has ended.
7. Mobile phones must be disconnected during the exam.
8. The exam will take **2,5 hours**.
9. **In this exam, the use of EDALib classes is NOT allowed. You cannot use interfaces and classes provided by the Java API (ArrayList, LinkedList, etc.). The use of Arrays is also forbidden. That is to say, in your solutions you must use your own data structures.**

DO NOT GO BEYOND THIS PAGE until the exam has started.

Problem 1 (2 points): Given the following classes:

```
public class SimpleNodeInt {

    public Integer elem;
    public SimpleNodeInt next;

    public SimpleNodeInt(Integer num) {
        elem = num;
    }
}

public class SimpleListInt {

    public SimpleNodeInt first;

}

public class DoubleNodeInt {

    public Integer elem;
    public DoubleNodeInt next;
    public DoubleNodeInt prev;

    public DoubleNodeInt(Integer num) {
        this.elem = num;
    }

}

public class DoubleListInt {

    public DoubleNodeInt header;
    public DoubleNodeInt trailer;

    public DoubleListInt() {
        header = new DoubleNodeInt(null);
        trailer = new DoubleNodeInt(null);
        header.sig = trailer;
        trailer.prev = header;
    }

}
```

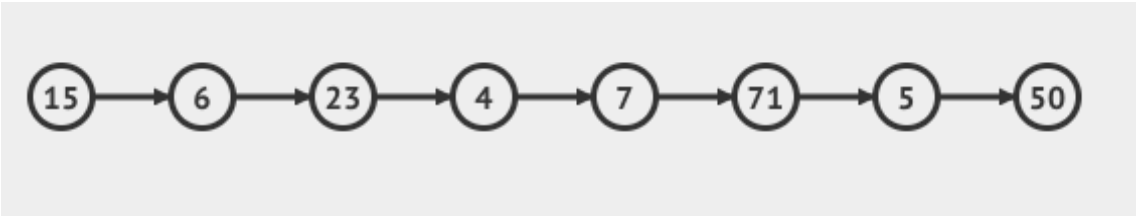
Implement a method called **searchandcopy,** which receives a simple list, a double list and an integer:

```
boolean searchAndCopy(SimpleListInt slist, DoubleListInt dlist, Integer elem)
```

searchandcopy method must search the **elem** value into the simple list **slist** and, if it is found, insert it in double list, **dlist,** in the same position it has in the original simple list **slist**. If the position of the integer value in the simple list is greater than or equal to the size of the double list, the number must be always inserted at the end of the double list. The method must return true if the number is found in the simple list and false otherwise.

Example: suppose that value 7 must be found:

Simple list:



The **elem** is at position 4 so, it must be stored in position 4 in the double list. Remember that positions in ADTs start at 0.

Double list:

Note:

- In a simple list there are no duplicates.
- The double list allows duplicates so checking if the value to be inserted is already in the list is not needed.
- Lists are not ordered.

solution:

```
boolean searchAndCopy(SimpleListInt slist, DoubleListInt dlist, Integer elem) {
    int pos=slist.searchPosition(elem);
    if (pos==-1) {
        System.out.println(elem + " was not found!!!.");
        return false;
    } else {
        System.out.println("Position of "+ elem + ": " +pos);
    }
    dlist.insertAt(elem, pos);
    return true;
}
```

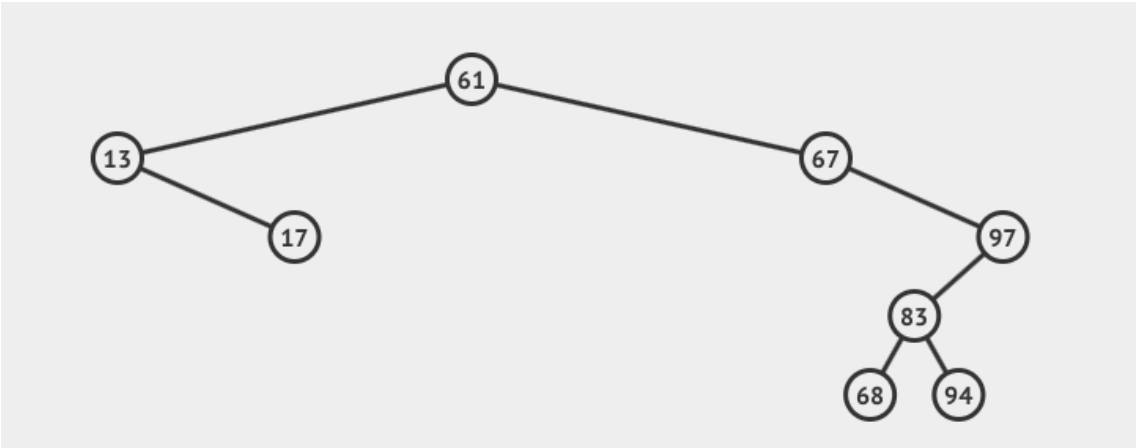
The method searchPosition should be in SimpleListInt:

```
/**
 * search the position of elem in the list.
 * If it is not found return -1
 * @param elem
 * @return
 */
public int searchPosition(int elem) {

    int pos=0;
    boolean found=false;
    SimpleNodeInt snodeIt = this.first;

    while (snodeIt != null && !found) {
        if (snodeIt.elem == elem) {
            found=true;
        }
        snodeIt = snodeIt.next;
        pos++;
    }

    if (!found) {
        pos=-1;
    }
    return pos;
}
```

The method inserAt should be in DoubleListInt

```
public void insertAt(int elem,int pos) {

    DoubleNodeInt nodeIt = this.header.next;
    int i=0;

    while (nodeIt != this.trailer && i<pos) {
        nodeIt = nodeIt.next;
        i++;
    }

    if (nodeIt==this.trailer) {
        System.out.println("position " + pos + " is greater "
                + "than the size of list (" + i +"). "
                + "The elem has to be added at the end of "
                + "the list.");
    }

    DoubleNodeInt newN = new DoubleNodeInt(elem);
    newN.next = nodeIt;
    newN.prev = nodeIt.prev;
    nodeIt.prev.next = newN;
    nodeIt.prev = newN;

}
```

Problem 2: Given the following binary search tree: **(Total 0,5 points)**:



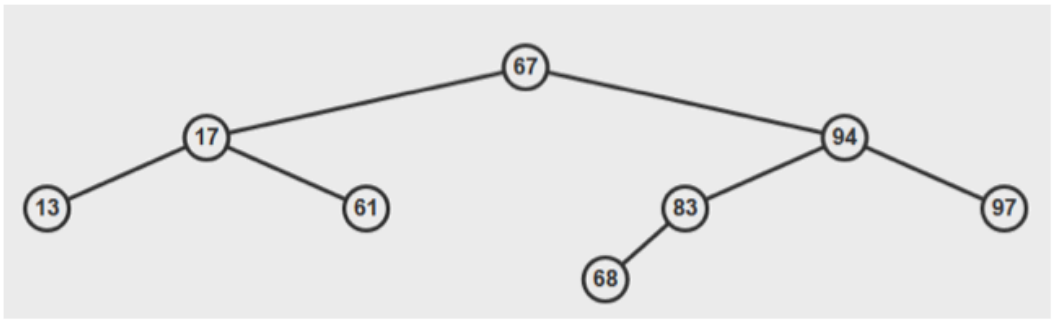
Answer the following:

a) Size balanced tree version (0.20)
b) High balanced version of the tree (AVL). (0.20)
c) What is the purpose of balancing a binary search tree? (0.1)

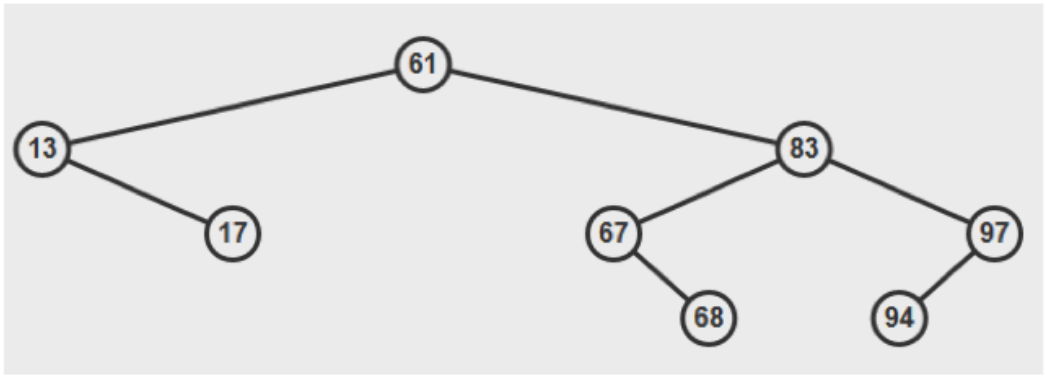
Note: The solution must include each of the steps followed to get the final result.

Solution:

a)



AVL



Problem 3 (**Total 3.5 points**)**:** Each hour, the control tower of the NiceFlights airport receives requests from the flights needing to take land in its runway (the airport has only one runway). The flight identifier (ex., IB1234) and the exact minute at which the runway is requested (ex., 35) must be somehow stored.

Answer the following:

a) (**0.5 points**) Which data structure do you suggest to store all requests to be sure thar search, insert and remove operations have logarithmic complexity?. Develop class(es) according to your answer for the previous question. In this section you ONLY have to write the class header, attributes and constructors (if you need them). So, you must not add any method.

b) (**1 point**) Write method **busyRunway**. This method receives an integer as input parameter, which represents the minute in which the activity in the runway must be checked. If the runway is booked, the method must return the identifier for the flight that made the reservation. In any other case, the method must return null. The method must have a recursive implementation, (**if you choose the iterative version, the maximum mark will be 0.5**).

c) (**1 point**) Write **receiveRequest** method. This method gets the flight identifier and the minute booked and stores the request. If the runway is already booked at that minute, the method will only warn about this situation. The method must have a recursive implementation, (**if you choose the iterative version, the maximum mark will be 0.5**).

d) (**1 point**) To avoid accidents, the coincidence of more than one plane in the runway must be avoided. The maximum time for a plane to be in the runway is 3 minutes. So, once the runway is booked, it is not possible to book it again in the following three minutes. For example, if the runway is booked at minute 25, it is not possible to make additional reservations in minutes 22,23,24,26,27,28 (of course, neither minute 25). Write a new version for the method **receiveRequest** avoiding crashes in the runway.

Solución:

a) Binary Search Treeproblema

```
public class Flight {

    public int minute;
    public String idFlight;

    public Flight parent, left, right;

    public Flight(int minuto, String vuelo) {
        this.minute = minuto;
        this.idFlight = vuelo;
    }

}
```

```
public class AirStrip {

    public Flight root;
```

b)

```
public String isTaken(int minuto) {
    return this.search(root, minuto);
}

private String search(Flight requestFlight, int minuto) {
    if (requestFlight != null) {
        // search in the left child
        if (requestFlight.minute > minuto)
            return search(requestFlight.right, minuto);
        // search in the right child
        if (requestFlight.minute < minuto)
            return search(requestFlight.left, minuto);
        // return the id flight
        return requestFlight.idFlight;
    } else
        return null;
}
```

c)

```java
public void saveRequest(int minuto, String vuelo) {
    if (isEmpty()) {
        root=new Flight(minuto,vuelo);
        return;
    }
    this.insert(root, minuto, vuelo);
}

private void insert(Flight requestFlight, int minuto, String vuelo) {
    if (requestFlight != null) {
        //the airtrip is taken
        if (requestFlight.minute == minuto) {
            System.out.println("Airtrip is already taken in " + minuto);
            return;
        }
        // busca por la derecha
        if (requestFlight.minute > minuto)
            if (requestFlight.right != null)
                insert(requestFlight.right, minuto, vuelo);
            else {
                requestFlight.right = new Flight(minuto, vuelo);
                requestFlight.right.parent = requestFlight;
            }
        if (requestFlight.minute < minuto)
            if (requestFlight.left != null)
                insert(requestFlight.left, minuto, vuelo);
            else {
                requestFlight.left = new Flight(minuto, vuelo);
                requestFlight.left.parent = requestFlight;
            }
    }
}
```

d)

```
public void saveRequest3(int minuto, String vuelo) {
    if (isEmpty()) {
        root=new Flight(minuto,vuelo);
        return;
    }
    this.insert3(root, minuto, vuelo);
}

private void insert3(Flight request, int minuto, String vuelo) {
    if (request != null) {
        if (request.minute + 3 > minuto && request.minute - 3 < minuto) {
            System.out.println("Airstrip is taken in "
                    + "(" + (request.minute - 3) + "-" + (request.minute + 3)+")");
            return;
        } else if (request.minute > minuto)
            if (request.right != null)
                insert3(request.right, minuto, vuelo);
            else {
                request.right = new Flight(minuto, vuelo);
                request.right.parent = request;
            }
        else if (request.minute < minuto)
            if (request.left != null)
                insert3(request.left, minuto, vuelo);
            else {
                request.left = new Flight(minuto, vuelo);
                request.left.parent = request;
            }
        }
}
```