Estructura de Datos y Algoritmos,
Grado en Ingeniería Informática, 1º,
Examen Final, 26 Mayo de 2017

Nombre:
Grupo:

Problema 1. (2 puntos)
Dada la interface IBSTree y las clases BSTNode y BSTree:

```
public interface IBSTree{
        public String find(Integer key);
        public void insert(Integer key, String elem);
        public void remove(Integer key);
}


public class BSTNode{
        Integer key;
        String elem;

        BSTNode parent;
        BSTNode left;
        BSTNode right;


        public BSTNode(Integer k, String e) {
                key = k;
                elem = e;
        }
…
}

public class BSTree implements IBSTree {
        BSTNode root;

        public String find(Integer key) {...}
        public void insert(Integer key, String elem){...}
        public void remove(Integer key){...}
}
```

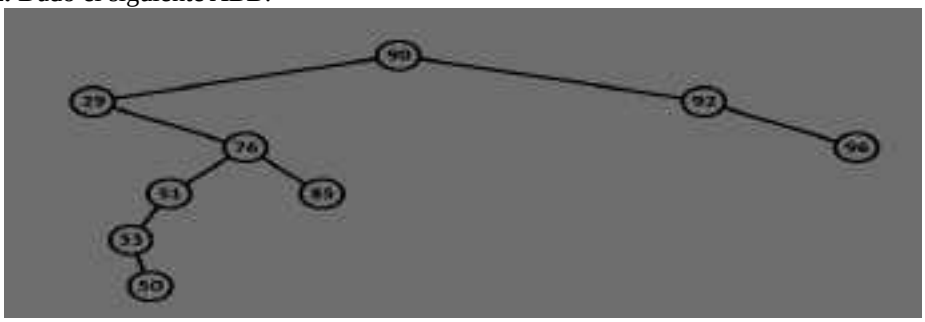
Implementa un **método recursivo** en la clase BSTree que borre el nodo con la clave más pequeña y devuelva dicha clave. En tu solución no puedes utlizar el método remove de la clase BSTree.

Solución:

```
public Integer removeMin() {
        if (root==null) {
                throw new NullPointerException("Tree is empty");
        }
        return removeMin(root);
}
```

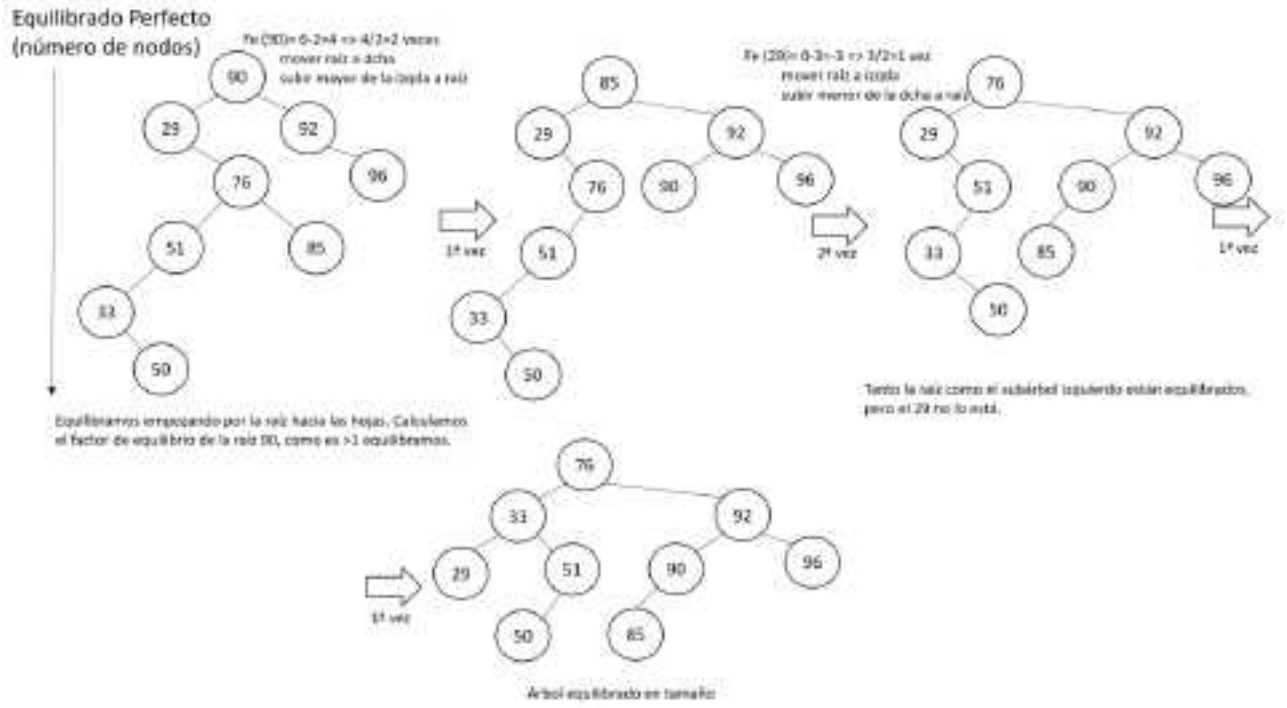
```
public Integer removeMin(BSTNode node) {
        if (node.left!=null) return removeMin(node.left);
        else {
                int key=node.key;
                //root=remove(key,root);
                //now, we have to remove nodo
                BSTNode parent=node.parent;
                if  (parent==null) {
                        root=nodo.right;
                } else {
                        //node came from the left branch of its parent.
                        //therefore, if we remove node, then node.right should be the
        //left child of its parent
                        parent.left=node.right;
                        if (node.right!=null) {
                          node.right.parent=parent;
                        }
                }
                return key;
        }
}
```

Problema 2. Dado el siguiente ABB:



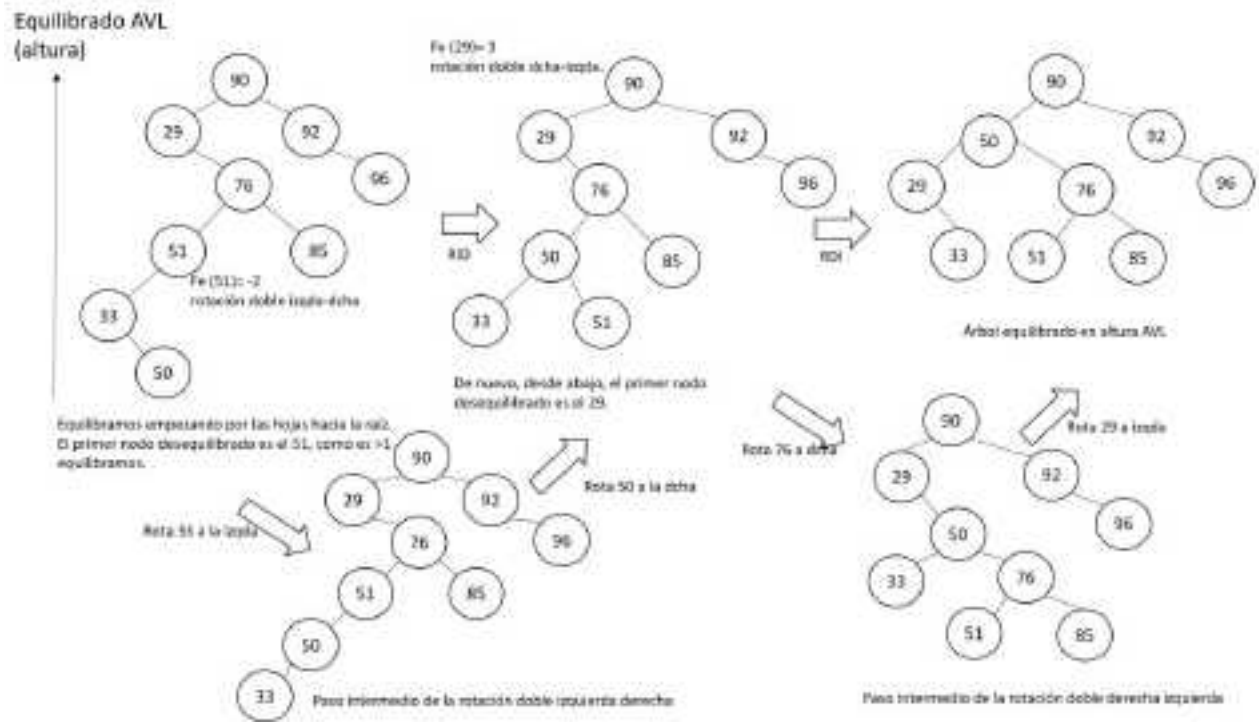
2.1. (0.2 ptos) Equilibra el árbol siguiendo un enfoque de equilibrado perfecto. Incluye en tu solución todos los árboles intermedios que obtienes tras aplicar cada uno de los pasos necesarios hasta converger a la solución final. Incluye una pequeña leyenda en cada árbol que describa la operación realizada para obtener dicho árbol.

Solución:



2.2. (0.2 ptos) Equilibra el árbol siguiendo un enfoque de equilibrado AVL. Incluye en tu solución todos los árboles intermedios que obtienes tras aplicar cada uno de los pasos necesarios hasta converger a la solución final. Incluye una pequeña leyenda en cada árbol que describa la operación realizada para obtener dicho árbol.

Solución:



2.3. (0.1 ptos)¿Cuál es el objetivo que se persigue el equilibrado de un árbol?

Solución: Mejorar las búsquedas ya que la complejidad de una búsqueda en un árbol equilibrado se mantiene en un orden logarítmico.

Problema 3 (2.5 ptos). Implementa un grafo para almacenar los números enteros del 0 al 99. La implementación de dicho grafo debe estar basada en listas de adyacencias (es decir, no debes usar una matriz de adyacencias). Las aristas del grafo representarán la relación **divisor_de** entre dos números. Aunque un número siempre es divisible por sí mismo, está relación no debe ser representada. Algunos ejemplos de aristas correctas son: (27,54), (27,81). Es decir, 27 es divisor de 54 y también de 81. Se pide implementar los siguiente métodos dentro de la clase del grafo:

3.1. (0.25) Escribe la cabecera de la clase GraphNumbers e incluye los atributos necesarios para representar un grafo basado en listas de adyacencias. Para la representación de las listas de adyacencias puedes suponer que ya cuentas con las siguientes clases implementadas:

```
public class DNodeVertex {

        public Integer vertex;

        public DNodeVertex prev;
        public DNodeVertex next;

        public DNodeVertex(Integer v) {
                vertex = v;
        }

}


public class DListVertex {

        DNodeVertex header;
        DNodeVertex trailer;
        int size=0;
```

```
        public DListVertex() {...}

        //puedes suponer que el método inserta al final de la lista
        public void add(int v){…}

        public boolean isEmpty(){…}

        public boolean contains(int v){…}

        public int getSize{…}

        public DNodeVertex getAt(int index) {…}

}
```

Solución:

```
public class GraphNumbers  {

        public static final int NUMVERTICES=100;
        DListVertex[] vertices=new DlistVertex[NUMVERTICES];
```

3.2. (0.5 ptos) Crea el método constructor que inicializa el grafo con los números del 0 al 99 y sus relaciones divisor_de. ¿Cuál es la complejidad del método?. ¿Cuál sería la complejidad del método constructor si el método add de DlistVertex almacenase los vértices de forma ordenada de menor a mayor?. Razona tu respuesta.

Solución:

```
        public GraphNumbers() {
                //creates each list
                for (int i=0; i<NUMVERTICES;i++) {
                        vertices[i]=new DListVertex();
                        if (i!=0) { //0 cannot be divisor
                                int multiplo=2*i;
                                while (multiplo<NUMVERTICES) {
                                        vertices[i].add(multiplo);//by default its weight is 0.
                                        multiplo=multiplo+i;
                                }
                        }
                }

        }
```

Complejidad $O(n^2)$ porque el algoritmo recorre el número de vértices y por cada uno de ellos, calcula sus posibles múltiplos y los añade a su lista de adyacencia. Si el método add sólo inserta al final de la lista, su complejidad en una lista doble, tendría complejidad constante. Por tanto, la complejdiad del método sería cuadrática, porque hay dos bucles anidados que siempre se recorren hasta alcanzar el número de vertices. Si el método add inserta en mitad de la lista (por ejemplo, de forma ordenada), su complejidad sería lineal, y por tanto, la complejidad sería $O(n^3)$.

3.2. (0.5 ptos) Implementa un método que reciba un número y que imprima por pantalla sus múltiplos comprendidos entre 0 y 99. El método debe utilizar la estructura de grafo para obtener estos múltiplos. Es decir, no se considerará una solución correcta si calculas directamente los múltiplos del número. ¿Cuál es la complejidad temporal de este método?. Razona tu respuesta.

Solución:

```
public void showMultiples(int i) {
                if (i<0 || i>=NUMVERTICES) {
                        System.out.println("Error: nonexistence vertex " + i);
                        return;
                }
                System.out.print("Multiples for " + i + ":");
                for (int j=0; j<vertices[i].getSize();j++) {
                        System.out.print(vertices[i].getAt(j).vertex+"\t");
```

```
        }
    }
```

Solución: El método tienen complejidad cuadrática porque la complejidad del bucle for se calcularía multiplicando el número de veces que se ejecuta (es decir el tamaño de la lista asociada al vértice i) por la complejidad del método getAt(). Dicho método de una lista tiene complejidad lineal.

3.3. (0.5) Implementa un método que reciba un número y que imprima por pantalla sus divisores. Recuerda que los números siempre deben estar comprendidos entre 0 y 99. Utiliza el grafo para obtener dichos divisores. Es decir, no se considera una solución correcta si calculas directamente los divisores del número. ¿Cuál es la complejidad del método?. Razona tu respuesta.

```
public void showDivisors(int i) {
        if (i<0 || i>=NUMVERTICES) {
            System.out.println("Error: nonexistence vertex " + i);
            return;
        }
        System.out.print("Divisors for " + i + ":");
        for (int j=0; j<NUMVERTICES;j++) {
            if (j!=i) {
                if (vertices[j].contains(i)) {
                    System.out.print(j+"\t");
                }
            }
        }
        System.out.println();
    }
```

Solución: El método tienen complejidad cuadrática porque la complejidad del bucle for se calcularía multiplicando el número de veces que se ejecuta (es decir el tamaño de la lista asociada al vértice i) por la complejidad del método contains(). Dicho método de una lista tiene complejidad lineal.

3.4. (0.75) Implementa un **método recursivo** que reciba un vértice e imprima por pantalla el recorrido en profundidad a partir de dicho vértice.

Solución:

```
    public void depth(int i)  {

        boolean[] visited= new boolean[NUMVERTICES];
        depth(i,visited);
    }


    protected void depth(int i,boolean[] visited) {
        //prints the vertex and marks as visited
        System.out.print(i+"\t");
        visited[i]=true;
        //gets its adjacent vertices
        int[] adjacents=getAdjacents(i);
        for (int adjV:adjacents) {
            if (!visited[adjV]) {
                //only depth traverses those adjacent vertices
                //that have not been visited yet
                depth(adjV,visited);

            }
        }
    }


    public int[] getAdjacents(int i) {
```

```
        if (i<0||i>=NUMVERTICES) {
                System.out.println("Nonexistent vertex  " + i);
                return null;
        }

        //gets the number of adjacent vertices
        int numAdj=vertices[i].getSize();
        //creates the array
        int[] adjVertices=new int[numAdj];
        //saves the adjacent vertices into the array
        for (int j=0; j<numAdj; j++) {
                adjVertices[j]=vertices[i].getAt(j).vertex;
        }
        //return the array with the adjacent vertices of i
        return adjVertices;
}
```

Problema 4 (1 pto)**.**
Utilizando la estrategia de divide y vencerás, implementa un método que reciba un array de enteros y devuelva la media de los elementos almacenados en el array. Recuerda que la solución debe seguir el enfoque de divide y vencerás, por tanto cualquier versión iterativa no será evaluada.

Solución:

```
//divide and conquer, obtains the average of the elements of the array
    public static double average (int[] A) {
        if (A==null || A.length==0) {
                System.out.println("Error: array is empty!!!");
                return -1;
        }

        double suma=sum(A, 0, A.length - 1);
        suma=suma/A.length;
   return  suma;
    }
    public static int sum(int[] A, int start, int end) {
          if (start>end || start<0 || end>a.length) return -1;
        if (start==end)  return A[start];

        int mid = (start+end)/2;
        int sumLeft = sum(A, start, mid);
        int sumRight = sum(A, mid+1, end);
        return sumLeft + sumRight;
    }
```

También puede hacerse:

```
    public static double average(int[] A) {
        if (A==null || A.length==0) {
                System.out.println("Error");
                return -1;
        }
        return average(A,0,A.length-1);
    }

    protected static double average(int[] A, int start, int end) {
        if (start>end || start<0 || end>A.length-1) {
                System.out.println("Error");
                return -1;
        }
        else {
                if (end == start)
                        return A[start];
```

```
                else {
                        int m = (end-start+1)/2; // length Left array
                        int n = end-start+1-m;  // slength Right array
                        double avgLeft = average (A, start, start+m-1);
                        double avgRight = average (A, start+m, end);
                        return (avgLeft*m + avgRight*n)/(m+n);
                }
        }
}
```