Estructura de Datos y Algoritmos,
Grado en Ingeniería Informática, 1º,
Examen Extraordinario, 20 de Junio de 2017

Nombre:
Grupo:

Problema 1. (2 puntos)
Dada la clase de **nodo binario de búsqueda** BSTNode:

```
public class BSTNode{
        public  Integer key;
        public  String elem;
        public  BSTNode parent;
        public BSTNode left;
        public BSTNode right;
        public BSTNode(Integer k, String e) {
                key = k;
                elem = e;
        }
…
}
```

a) (0.5 ptos) Implementa un **método recursivo** en la clase BSTNode que reciba un nodo y devuelva su tamaño (es decir, el tamaño del subárbol que cuelga de dicho nodo).
Solución:

```
public int getSize() {
      return getSize(this);
}

private static int getSize(BSTNode node) {
      if (node == null)
            return 0;
      else
            return 1 + getSize(node.left) + getSize(node.right);
}
```

b) (0.5 ptos) Implementa un método en la clase BSTNode que reciba un nodo y calcule su factor de equilibrio en tamaño.

Solución:

```
public int getFactorSize() {
      return getFactorSize(this);
}
```

```
private static int getFactorSize(BSTNode node) {
	if (node == null)
		return 0;
	else
		return Math.abs(getSize(node.right) - getSize(node.left));
}
```

c) (0.5 ptos) Implementa un **método recursivo** en la clase BSTNode que reciba un nodo y que devuelva true si el subárbol que cuelga del nodo está equilibrado en tamaño y false en otro caso. Si el nodo es nulo, podemos considerar que está equilibrado.

Solución:

```
public boolean isBalancedSize() {
		return isBalancedSize(this);
}

private static boolean isBalancedSize(BSTNode node) {
	if (node == null)
		return true;
	else
		return getFactorSize(node)<=1 &&
		isBalancedSize(node.left) &&
		isBalancedSize(node.right);
}
```

d) (0.25 ptos) Implementa un método en la clase BSTNode que reciba un nodo y que devuelva su nodo sucesor (dentro del subárbol que cuelga del nodo). Si no tiene nodo sucesor deberá devolver null.

Solución:

```
public BSTNode getSucessor() {
		BSTNode succesor=this.right;
		if (succesor!=null) {
			while (succesor.left!=null) {
				succesor=succesor.left;
			}
		}
		return succesor;
	}
```

e) (0.25 ptos) Implementa un en la clase BSTNode que reciba un nodo y que devuelva su nodo predecesor (dentro del subárbol que cuelga del nodo). Si no tiene nodo predecesor deberá devolver null.

Solución:

```
public BSTNode getPredecessor() {
		BSTNode predecessor=this.left;
		if (predecessor!=null) {
			while (predecessor.right!=null) {
				predecessor=predecessor.right;
			}
		}
		return predecessor;
	}
```

Problema 2. (2 puntos)
Supón que en la anterior clase BSTNode además de los métodos implementados en el problema anterior, también dispones de los siguientes métodos ya implementados:
•**public static** String find(Integer key, BSTNode node){...}, que busca la clave key en el subárbol que cuelga del nodo node y devuelve su valor asociado o bien null si key no se encuentra en dicho subárbol.
•**public static BSTNode** insert(Integer key, String elem, BSTNode node){...}, que crea un nodo con clave key y valor elem y lo inserta dentro del subárbol que cuelga del nodo node. Devuelve el node node actualizado tras la inserción.
•**public static BSTNode** remove(Integer key, BSTNode node) {…}, que borra el nodo con clave key dentro del subárbol que cuelga del nodo node. Devuelve el nodo remove tras la eliminación del nodo con esa clave.
Implementa un método que reciba un nodo y equilibre dicho nodo en tamaño.

Solución:

```
public static BSTNode balance(BSTNode node) {
	int fe=node.getFactorSize();
	if (fe>1) {
		//we have to move fe/2 nodes
		for (int i=1; i<=fe/2; i++) {
		//while (!node.isBalancedSize()) {

			//move from left to right
			if (node.left.getSize()>node.right.getSize()) {
				BSTNode predecessor=node.getPredecessor();
				//move the node ot its right subtree
				node.right=insert(node.key, node.elem, node.right);
				//replace values for node by the predecessor's values
				node.key=predecessor.key;
				node.elem=predecessor.elem;
				//remove predecessor from its left child
				node.left=remove(predecessor.key,node.left);
			} else { //move from right to left
				BSTNode sucessor=node.getPredecessor();
				//move the node to its left subtree
				node.left=insert(node.key, node.elem, node.left);
				//replace values for node by the sucessor values
				node.key=sucessor.key;
				node.elem=sucessor.elem;
				//remove predecessor from its right child
				node.right=remove(sucessor.key,node.right);

			}
		}
	}
	//now, we have to balance the left and right subtrees
	if (node!=null) {
		node.left=balance(node.left);
		node.right=balance(node.right);
	}
	return node;
}
```

Problema 3 (2 ptos). Dada la interfaz de lista IList y las clases DNode y DList de números enteros (puedes suponer que los métodos de la interfaz están ya implementados), se pide implementar un método **public void** insertRevAt(**int** index, **int** newElem), que inserta el elemento newElem en la posición index pero empezando por el final de la lista. Es decir, si por ejemplo, index=0, entonces el elemento se tendrá que insertar justo al final de la lista (pero recuerda que siempre antes del nodo centinela trailer!!!); si index=1, el elemento se tendrá que insertar en la penúltima posición; si index=getSize(), entonces el elemento se tendrá que insertar como el primer elemento de la lista. En tu solución, no puedes utilizar el método insertAt de la interfaz IList, es decir, tu método no puede llamar al método insertAt. ¿Cuál es la complejidad de este método?. Escribe y razona brevemente sobre la complejidad de cada uno de los métodos de la clase DList.

Solución:

```
public void inserRevAt(int index, String elem) {
            DNode newNode = new DNode(elem);
            int i = 0;
            boolean inserted=false;
            for (DNode nodeIt = trailer; nodeIt != header && inserted==false;
nodeIt = nodeIt.prev) {
                  if (i == index) {
                        newNode.prev = nodeIt.prev;
                        newNode.next= nodeIt;
                        nodeIt.prev.next= newNode;
                        nodeIt.prev = newNode;
                        inserted=true;
                        size++;
                  }
                  ++i;
            }
            if (!inserted) System.out.println("DList: Insertion out of
bounds");
}
```

Nota: Si el problema permitiese el uso del método insertAt, la solución podría ser la siguiente:

```
public void inserRevAt(int index, String elem) {
      insertAt(size-index,elem);
}
```

La complejidad del método es lineal porque en el peor de los casos deberemos recorrer toda la lista para insertar el elemento.

Método	Complejidad
isEmpty()	O(1) porque el número de instrucciones siempre es constante
getSize()	
addFirst(..)	
addLast(..)	
removeFirst()	
removeLast()	
getFirst()	

getLast()	
insertAt(..)	O(n) porque en el peor de los casos es necesario recorrer toda la lista para realizar cada una de las operaciones. Por ejemplo, en el método contains() o getIndexOf(), en el peor de los casos (no existe) es necesario recorrer toda la lista para poder devolver false. En el método toString() siempre hay que recorrer toda la lista. En el método getAt o removeAt() en el peor de los casos, la posición consultada será la última.
contains(..)	
getAt(..)	
toString(..)	
getIndexOf(..)	
removeAt(..)	
removeAll(..)	

Problema 4 (3 puntos). Supón que Metro de Madrid quiere contratarnos para que desarrollemos una aplicación que les permita gestionar la red de sus estaciones. Alguna de la información que nos proporciona es la siguiente:

- El número de estaciones suele ser un número fijo y variar muy poco en el tiempo. Actualmente, el número de estaciones en el metro de Madrid son 301.
- De cada estación es necesario conocer el nombre.
- Además, una estación puede estar cerrada temporalmente por obras o por otras causas. Por tanto, es interesante que también almacenes de alguna forma su estado (por simplificar, sólo vamos a considerar dos estados: abierto o cerrado).

Se pide implementar una estructura de datos que permita representar dicha red. En concreto se solicita:

a) (0.25 ptos) Escribir los atributos de la estructura necesaria para representar la red de estaciones. Razona brevemente la elección de dichos atributos para representar la estructura.

Solución:

Para representar un estación, es necesario almacenar su nombre, su estado (abierta o cerrado; cuando se crea una estación vamos a suponer que su estado es true) y la lista de estaciones con las que tiene conexión.

```
public class Station {
	//station's name
	public String name;
	//true if the station is open, false eoc
	public boolean isopen;
	//list to store the linked stations to this station
	public DListStations lst= new DListStations();

	public Station(String n) {
		name=n;
		isopen=true;
		lst=new DListStations();
	}

	public void close() {
		isopen=false;
	}
	public void open() {
		isopen=false;
	}
```

```
	...

}
```

La lista para almacenar las estaciones con las que está conectada una determinada estación puede ser implementada como una lista doble de nodos, donde en cada nodo se almacene el nombre de la estación (otra opción es almacenar el índice del array donde está almacenada dicha estación) y un valor float para almacenar la distancia en kilómetros entre estaciones.

```
public class DListStations implements IList {

	DNodeStation header;
	DNodeStation trailer;
	int size=0;

	public DListStations() {
		header = new DNodeStation(null,null);
		trailer = new DNodeStation(null,null);
		header.next = trailer;
		trailer.prev= header;
	}
	…
}
```

Para la clase que representa los nodos de las estaciones conectadas en la lista, hemos optado por en lugar de guardar el nombre de la estación, vamos almacenar el índice del array en el que está almacenada la estación en concreto. Esto nos permite un acceso más directo a la hora de manipular la lista, pero podría hacerse igualmente almacenando el nombre de la estación directamente

```
public class DNodeStation {

	//the index of the station in the array of stations
	public Integer indexOfStation;
	//distance between the two stations
	public Float distance;

	public DNodeStation prev;
	public DNodeStation next;

	public DNodeStation(Integer v, Float w) {
		indexOfStation = v;
		distance = w;
	}

}
```

Para representar la red de estaciones del Metro, bastaría con declarar un array de objetos Station..

```
public class Metro {
	public int NUMSTATIONS;
	public Station[] stations;
	…
```

}

b) (0.25 ptos) La estructura debe tener un método que permita inicializar la red de estaciones de metro. Dicho método podría recibir un array con los nombres de las estaciones. ¿Qué complejidad tiene este método?.

Solución:

```
public Metro(String names[]) {
   NUMSTATIONS=names.length;

   stations=new Station[NUMSTATIONS];
   for (int i=0; i<NUMSTATIONS;i++) {
         stations[i]=new Station(names[i]);
   }
}
```

La complejidad del método es linea (O(n)) porque tiene que siempre tiene que recorrer todo el array con los nombres de las estaciones. La complejidad del constructor Station es O(1).

c) (0.5 ptos) La estructura debe proporcionar un método que permita añadir conexiones entre dos estaciones. Para cada conexión se almacenará su longitud en kilómetros. Calcula la función tiempo de ejecución y el orden de complejidad de dicho método.

Solución:

```
public void addEdge(int i, int j, float d) {
      if (!checkStation(i)) //2
            throw new IllegalArgumentException("Nonexistent station  " + i);
      if (!checkStation(j)) //2
            throw new IllegalArgumentException("Nonexistent station  " + j);

      //if the station already exists into the list of i, then, we only have to
      //modify the distance between the two stations
      if (stations[i].lst.contains(j)) { //n

            int index=stations[i].lst.getIndexOf(j); //n
            DNodeStation node=stations[i].lst.getAt(index); //n
            node.distance=d;  //1

      //metro is a non-directed graph because their relations are symmetrical
            index=stations[j].lst.getIndexOf(i); //n
            node=stations[j].lst.getAt(index); //n
            node.distance=d; //1

      } else {
            stations[i].lst.addLast(j,d); //1
            stations[j].lst.addLast(i,d); //1
      }
}
```

El peor de los casos será cuando las estaciones ya están conectadas, por tanto, la función temporal del método será T(n)= 4 + 5n + 2 = 5n + 6
El mejor de los casos será cuando las estaciones no están conectadas. Su función temporal, será

T(n)=4 + n + 2 = n + 6. (n porque en cualquier caso tienes que ejecutar le método contains). La complejidad del método será lineal O(n).

d) **(0.5 ptos)** La estructura debe proporcionar un método que, dada una estación, devuelva un array con el nombre de las estaciones con las que tiene una conexión directa. Calcula la función tiempo de ejecución y el orden de complejidad de dicho método.

Solución:

```
public String[] getConections(String name) {

		int index=getIndexOfStation(name); //n
		if (index==-1) { //1
			System.out.println(name + " is not found!!!"); //1
			return null; //1
		} else {
			Station objStation=stations[index]; //1
			String[] links=new String[objStation.lst.size]; //1
			for (int i=0; i<objStation.lst.size;i++) {//1 + n+1+n
				int j=objStation.lst.getAt(i).indexOfStation; //n
				String n=getStationAt(j); //n
				links[i]=n; //1
			}

			return links; //1
		}

	}
```

En el peor de los casos (cuando el nombre de la estación está en el array), la función temporal será: T(n)= n + 1 + 1 +1 + 1 + n+1 + n + n*(n+n+1) + 1 = $2n^2$ + 4n + 6. Por tanto, la complejidad del método es cuadrática.

e) **(1.5 ptos)** La estructura debe proporcionar un método que, dada una estación de origen y una estación de destino, devuelva true si existe un camino entre ambas estaciones y false en otro caso. Para ello deberás utilizar el algoritmo de recorrido en amplitud de un nodo. Además, tendrás que tener en cuenta cuándo una estación está abierta o cerrada, ya que en caso de estar cerrada, el algoritmo no deberá continuar por ninguno de los caminos que salen de dicha estación.

Solución:

```
public boolean isConnected(String station1, String station2) {
		int index1=getIndexOfStation(station1);
		int index2=getIndexOfStation(station2);
		if (index1==-1) {
			System.out.println(station1 + " does not exist!!!");
			return false;
		}
		if (index2==-1) {
			System.out.println(station2 + " does not exist!!!");
			return false;
```

```
            }
            if (!stations[index1].isopen ||!stations[index2].isopen) {
                  System.out.println("stations should be open!!!");
                  return false;
            }
            boolean visited[]=new boolean[stations.length];



            SQueue q=new SQueue();
            q.enqueue(index1);

            boolean connected=false;

            while (!q.isEmpty() && !connected) {
                  int current=q.dequeue();
                  visited[current]=true;
                  if (current==index2) {
                        //we find it!!!
                        connected=true;
                  }
                  int directs[]=getConnections(current);
                  for (int adj:directs) {
                        if (stations[adj].isopen && !visited[adj] &&
                        !q.contains(adj))       q.enqueue(adj);

                  }
            }

            return connected;

      }
      //similar to getAdjacents
public int[] getConnections(int index) {
      if (!checkStation(index))
            throw new IllegalArgumentException("Nonexistent station  " +index);
      Station obj=stations[index];
      int[] links=new int[obj.lst.size];
      for (int i=0; i<obj.lst.size;i++) {
            int j=obj.lst.getAt(i).indexOfStation;
            links[i]=j;
      }
      return links;
}
```

Nota: El uso de matrices para la implementación del problema 4 será penalizado. En caso de usar matrices, los apartados anteriores sólo valdrán la mitad. Sin embargo, sí puedes suponer que tienes la clase DNode y DList del problema anterior implementadas y puedes modificarlas, si tu implementación así lo requiere.

Problema 5 (1 punto). Utilizando la estrategia de divide y vencerás, implementa un método que reciba un array de enteros y devuelva el número par más pequeño del array. Si un array no tiene ningún número par, el método devolverá -1. Recuerda que la solución debe seguir el enfoque de

divide y vencerás, por lo que cualquier otra versión recursiva que no siga este enfoque o iterativa no será evaluada.

Solución:

```
public static int findMinPair(int a[]) {
            if (a==null || a.length==0) {
                  System.out.println("Error: array is empty!!!");
                  return -1;
            }
            return findMinPair(a,0,a.length-1);
      }


private static int findMinPair(int a[], int start, int end) {

      //check if the indexes are right
      if (start>end || start<0 || end>a.length-1) {
            System.out.println("IndexError: out of index range!!!");
            return -1;
      }

      if (start==end) { //base case
            if (a[start]%2==0) return a[start];
            else return -1;
      } else { //general case
            int m=(start+end)/2;
            //divide
            int min1=findMinPair(a,start,m);
            int min2=findMinPair(a,m+1,end);
            //solve each subproblem
            if (min1==-1) return min2;
            if (min2==-1) return min1;

            return (min1<min2)?min1:min2; //combine

      }

      }
```