# Examen de Estructuras de Datos y Algoritmos
# (Ingeniería Informática)

### Septiembre 2009

## Primera parte (50% nota del examen)

1) Se desea hacer un método con la cabecera que se muestra abajo que cree y retorne un mapa que relacione elementos de la clase `E` con el número de apariciones de ese elemento en la lista de listas que se pasa como parámetro. Por cada elemento de las listas que no esté aún en el mapa, hay que meter en él ese elemento con un número de apariciones igual a la unidad. Por cada elemento de las listas que ya esté en el mapa, hay que incrementar en una unidad el número de apariciones.

```
public static <E> Map<E,Integer> numApariciones
      (List<List<E>> listaDeListas)
```

Indicar la eficiencia de esta operación.

2) Se desea crear una clase que permita almacenar una secuencia de elementos, posiblemente repetidos, de manera que sea eficiente comprobar si un elemento pertenece a la lista, y que también se conserve el orden en la lista. Para ello se utiliza una doble estructura de datos con un conjunto "hash" y una lista enlazada, como se muestra en el código de la clase.

```
public class ListaHash<E>
{
    private Set<E> conj= new HashSet<E>();
    private List<E> lista=new LinkedList<E>();

    public void inserta(E e) {...}

    public boolean borra(E e) {...}
}
```

Se pide implementar los métodos `inserta` y `borra`. El método `inserta` añade el nuevo elemento al final de la lista y lo añade también al conjunto. El método `borra` mira en el conjunto a ver si el elemento está almacenado. En caso afirmativo quita el elemento de la lista, si no quedan más elementos iguales también lo quita del conjunto, y retorna `true`. En caso negativo retorna `false`.

Indicar la eficiencia de estas dos operaciones.

¿Se te ocurre una forma de mejorar la eficiencia?

3) Se dispone del método `existeArista` que permite comprobar si existe en el grafo `g` una arista que va desde el vértice `origen` al `destino`.

Utilizar este método para escribir el pseudocódigo del método `estaIncluido` que comprueba si un grafo `g1` está contenido o no en otro grafo `g2`. Un grafo está contenido en otro si todos sus vértices están incluidos en él, y todas sus aristas existen en él.

método<E,A> existeArista(E origen, E destino, Grafo<E,A> g) retorna booleano

método <E,A> estaIncluido(Grafo<E,A> g1, Grafo<E,A> g2) retorna booleano

Observar que ambos métodos son genéricos. `E` es la clase de elementos guardados en los vértices del grafo, y `A` el contenido de las aristas.

# Examen de Estructuras de Datos y Algoritmos
# (Ingeniería Informática)

### Septiembre 2009

## Segunda parte (50% nota del examen)

4) Se dispone de la siguiente implementación de un mapa abierto (que se muestra sólo en parte). Se desea hacer la implementación de la operación `putAll` que copia todas las relaciones del mapa `t` al mapa almacenado en el objeto actual.

```
public class MapaAbierto<K,V>
{
    // atributos privados
    private ListaEnlazadaSimple<Entry<K,V>>[] tabla;

    private static class Entry<K,V> {
        private K clave;
        private V valor;

        public Entry(K clave, V valor) {
            this.clave=clave;
            this.valor=valor;
        }

        public boolean equals(Object otra) {...}
    }

    private int valorHash(K clave) {
        return clave.hashCode() % tabla.length;
    }

    /**
     * Asigna un valor a una clave
     */
    public V put(K clave, V valor) {...}

    /**
     * Copia todas las relaciones del mapa t al mapa almacenado
     * en el objeto actual
     */
    public void putAll(MapaAbierto<K,V> t) {...}

}
```

5) Se desea implementar un conjunto que implementa la interfaz `Set` de las colecciones Java usando un árbol binario ordenado para conseguir una buena eficiencia en el peor caso (O(log n)). Abajo se muestra parte de la implementación. Se pide escribir el método `next()` del iterador.

   Para recorrer en orden un árbol binario ordenado, observar que hay que empezar por el elemento situado más a la izquierda, que es el menor. Esto lo hace el constructor del

iterador empezando por la raíz y luego llamando al método `desciende` para bajar al descendiente situado más a la izquierda.

Para avanzar al siguiente elemento con `next`, si hay hijo derecho se baja a él y luego se llama al método `desciende` para ir al descendiente situado más a la izquierda. Si no hay hijo derecho entonces hay que subir al padre. Si el contenido del padre es mayor que el del hijo del que venimos, nos quedamos en él. Si es menor, hay que ir ascender al padre de él, y así sucesivamente. De esta manera se recorren los nudos en orden de sus valores.

El método `next` debe retornar el contenido que tenía el iterador llamado `actual` antes de moverlo.

```
import java.util.*;
import adts.*;
public class ConjuntoArbol<E extends Comparable> extends AbstractSet<E>
{
    private ArbolBinario<E> arbol; // el árbol binario
    private int num=0; // número de elementos

    public static class Iterador<E extends Comparable>
        implements Iterator<E>
    {
        // atributos del iterador
        private IterArbolBin<E> actual;

        // método privado que lleva el iterador hacia
        // el descendiente más a la izquierda
        private void desciende() {
            try {
                while (actual.tieneHijoIzquierdo()) {
                    actual.irAHijoIzquierdo();
                }
            } catch (NoValido e) {
                // No puede darse
            }
        }

        public Iterador(ConjuntoArbol<E> conj) {
            actual = conj.arbol.iterador();
            desciende();
        }

        public boolean hasNext() {... }

        public E next() {...}
    }
}
```

6) Se dispone de una implementación de una lista enlazada simple como la vista en clase. Escribir el método `divide()`, cuya cabecera se indica abajo, y que sirve para partir la lista por la mitad. La primera mitad contendrá los elementos del 0 al (`num`-1)/2 (división entera, siendo `num` el tamaño de la lista) y se quedará en el objeto actual. La 2ª mitad contendrá el resto de los elementos y será retornada como una nueva lista por el método. Hacer también un diagrama con un ejemplo de la lista antes de llamar al método, y cómo quedan las dos listas después de la llamada.

```
import java.util.*;
public class ListaEnlazadaSimple<E>
   extends AbstractSequentialList<E>
{
    // atributos privados
    private Celda<E> principio;
    private int num;

    // clase privada que define la celda
    private static class Celda<E> {
       E contenido;
       Celda<E> siguiente;

       Celda(E cont) {
          contenido=cont;
       }
    }

    /**
     * Constructor que crea la lista vacia
     */
    public ListaEnlazadaSimple()
    {
        // crea la celda cabecera
        principio=new Celda<E>(null);
        num=0;
    }

    /**
     * Divide la lista en dos
     */
    public ListaEnlazadaSimple<E> divide() {
        return new ListaEnlazadaSimple<E>();
    }
    ...
}
```