ESTRUCTURA DE DATOS Y ALGORITMOS **17-Septiembre-2007**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

1. Ejercicio (1 punto)

Dado el array A={8, 3, 7, 1, 4, 9, 5, 2, 6}

Describir cual es el estado del array A después de cada paso principal del algoritmo: `InsertionSort`.

{8, 3, 7, 1, 4, 9, 5, 2, 6}→{3, 8, 7, 1, 4, 9, 5, 2, 6}→{3, 7, 8, 1, 4, 9, 5, 2, 6}→{1, 3, 7, 8, 4, 9, 5, 2, 6} →{1, 3, 4, 7, 8, 9, 5, 2, 6}→{1, 3, 4, 7, 8, 9, 5, 2, 6}→{1, 3, 4, 5, 7, 8, 9, 2, 6}→{1, 2, 3, 4, 5, 7, 8, 9, 6} →{1, 2, 3, 4, 5, 6, 7, 8, 9}

2. Ejercicio. (2,5 puntos)

Supongamos una lista cuyos elementos pueden ser números o a su vez otras listas. Definir el método `obtenerOcurrencias`, que permita conocer cuántas veces aparece un determinado número en total en todas las listas.
Se pide:

a) (0,5) **Define** las clases necesarias para definir la estructura de una lista enlazada.

```
class Node {
   Object element;
   Node next;
}
public class LinkedList{
   Node top;
}
```

```
public class LinkedListItr {
   LinkedList theList;
   Node current;
   Node preceding;
}
```

b) (1,5) **Implementa** en Java el método `obtenerOcurrencias` y los métodos de la lista enlazada que hayas utilizado.

```
Por un lado, en la lista se almacenan objetos de la clase Integer o
LinkedListItr (o LinkedList). Por otro lado, en cuanto a la
codificación del método se me ocurren dos soluciones:
1) El método se implementa en una librería genérica, por ello la lista
a analizar se pasaría como parámetro:
   public static int obtenerOcurrencias (LinkedListItr lista,
                                         int numero)

2) El método se implementa en la clase iteradora de la lista.

public int obtenerOcurrencias (int numero){
  int ocurrencias = 0;
  goFirst();
  while (isOnList()){
    Object elemento = getCurrent();
    if (elemento instanceof LinkedListItr)
      ocurrencias = ocurrencias +
                    ((LinkedListItr)element).obtieneOcurrencias(numero);
    elseif (((Integer)elemento).intValue() == numero)
         ocurrencias++;
    advance();
  }
  return ocurrencias;
}
```

3. Ejercicio (2,5 puntos)

Especificar, diseñar e implementar un método (de la clase `BinTreeItr`) de un árbol binario de números enteros, que dado un determinado número entero devuelva el número de nodos del árbol que tienen un valor menor que el número dado.

1. Especificar / Parametrizar

Entrada: *A*, un arbol binario y un número entero, *n*
Salida: un número entero, que representa el número de nodos que tienen un valor menor a *n*
Llamada inicial: `numNodos`(A, n)

2. Diseño

Casos triviales

Si Vacio (A) → 0

Caso General

Si no_Vacio (A) y Raíz(A) < n → 1 + `numNodos` (Izq(A), n) + `numNodos` (Drch(A), n)
Si no_Vacio (A) y Raíz(A) >= n → `numNodos` (Izq(A), n) + `numNodos` (Drch(A), n)

3. Implementación

Los siguientes métodos se implementarían en la clase BinTreeItr:

```
int numNodos (int n ){
   return numNodos(bTree.root, n)
}

int numNodos(BTNode nodo, int n ){
   int cont;
   if (nodo == null) return 0;
   else {
      cont = numNodos(nodo.hijoIz(), n) + numNodos(nodo.hijoDr(), n);
     if (nodo.raiz() < n )
       cont = cont + 1;
   }
   return cont;
}
```

4. Ejercicio (4 puntos)

Desarrollar un sistema que modele el comportamiento de un despacho de abogados. Para ello suponemos disponibles las siguientes clases y su implementación en Java, todas ellas equipadas con las operaciones de igualdad (i.e. `equals`) y orden (i.e. `compareTo`):

- `Abogado` que sirve para almacenar y gestionar la información sobre un abogado, por ejemplo, dni, nombre, dirección, …
- `Cliente` que sirve para almacenar y gestionar la información sobre un cliente.

Se nos pide implementar la clase `Despacho`, cuya misión es gestionar la información sobre los abogados y los clientes. Esta clase ofrece las siguientes operaciones públicas:

public Despacho(): Genera un despacho vacío sin ninguna información.
public void nuevoAbogado(Abogado abogado): Altera un despacho, dando de alta un nuevo abogado que antes no figuraba en el despacho.
public void pideCita(Abogado abogado, Cliente cliente): Altera un despacho, haciendo que un cliente se ponga a la espera para ser atendido por un abogado, el cual debe estar dado de alta en el despacho.
public Cliente siguienteCliente(Abogado abogado): Devuelve el cliente a quien le toca el turno para ser atendido por un abogado; éste debe estar dado de alta y debe tener algún cliente que le haya pedido cita.
public void atiendeCliente(Abogado abogado): Modifica un despacho, eliminando el cliente al que le toque el turno para ser atendido por un abogado; éste debe estar dado de alta y debe tener algún cliente que le haya pedido cita.
public boolean tieneCliente(Abogado abogado): Reconoce si hay o no clientes a la espera de ser atendidos por un abogado, el cual debe estar dado de alta.

Se pide:
a.) **Desarrollar** en Java una implementación de la clase `Despacho` basada en otras estructuras de datos conocidas, **optimizando la complejidad** temporal de las operaciones. **Argumenta**, brevemente, la forma de estructura que has seleccionado para organizar la información del despacho. Se debe implementar cada una de las operaciones indicadas en base a la estructura seleccionada, **controlando los posibles errores** que se puedan ocasionar.

Nota: Si necesitas cualquier otra clase, aparte de las estructuras de datos conocidas, impleméntala.

Aclaración de los nombres sinónimos de los métodos de la clase Hashtable utilizados en la implementación y vistas en clase:

```
- containsKey → exists
- get → find
- put → insert y/o modidfy
```

```
class AbogadoClientes implements Entry{
  // El objetivo de esta clase es relacionar un abogado con una cola de clientes que están esperando ser
  // atendidos por él. Además, implementará la interfaz Entry para facilitar que está información  sea
  // almacenada en una Hashtbale
    private Abogado _abogado;
    private Queue _clientesEnEspera;  // esta cola almacena objetos de tipo Cliente

    public AbogadoClientes(Abogado abogado){
        _abogado = abogado;
        _clientesEnEspera = new Queue();
    }
```

```
   public Objetc getKey(){
      return _abogado.getDni();
   }

   public Object getObject(){
      return _clientesEnEspera;
   }

   public Object setObject(Object clientesEnEspera){
      If (clientesEnEspera instanceof Queue)
         _clientesEnEspera = clientesEnEspera;
   }
}

class EAbogadoInexistente { };
class EAbogadoRepetido { };
class EAbogadoSinClientes { };

class Despacho{
   private Hashtable _despacho; // En la hashtable guardaremos instancias de la clase
                  // AbogadoClientes que continen la relación de un abogado y la lista de clientes que
                  // están esperando ser atendidos.

   public Despacho(){
      _despacho = new Hashtable();
   }

   public void nuevoAbogado(Abogado abogado) throw EAbogadoRepetido{
       AbogadoClientes ac = new AbogadoClientes(abogado);
      if (_despacho.contains(ac)) throws new EAbogadoRepetido();
      _despacho.put(ac.getKey(),ac.getObject());
  }

  public   void   pideCita(Abogado   abogado,   Cliente   cliente)   throw
EAbogadoInexistente {
     AbogadoClientes ac = new AbogadoClientes(abogado);
     if ( !_despacho.contains(ac) ) throws EAbogadoInexistente();
     Queue colaClientesEnEspera = (Queue) _despacho.get(ac.getKey());
     colaClientesEnEspera.enqueue(cliente);
  }

  public Cliente siguienteCliente(Abogado abogado) throw EAbogadoInexistente,
EAbogadoSinClientes{
    AbogadoClientes ac = new AbogadoClientes(abogado);
    if ( !_despacho.contains(ac) ) throws EAbogadoInexistente();
    Queue colaClientesEnEspera = (Queue) _despacho.get(ac.getKey());
    if (colaClientesEnEspera.isEmpty() ) throws EAbogadoSinClientes();
    return colaClientesEnEspera.first();
  }
```

```
  public void atiendeCliente(Abogado abogado) throw EAbogadoInexistente,
EAbogadoSinClientes{
     AbogadoClientes ac = new AbogadoClientes(abogado);
     if ( !_despacho.contains(ac) ) throws EAbogadoInexistente();
     Queue colaClientesEnEspera = (Queue) _despacho.get(ac.getKey());
     if (colaClientesEnEspera.isEmpty() ) throws EAbogadoSinClientes();
    colaClientesEnEspera.dequeue();
    _despacho.put(abogado, cola);
  }

  public boolean tieneCliente(Abogado abogado) throw EAbogadoInexistente{
    AbogadoClientes ac = new AbogadoClientes(abogado);
    if ( !_despacho.contains(ac) ) throws EAbogadoInexistente();
    Queue colaClientesEnEspera = (Queue) _despacho.get(ac.getKey());
    return !(colaClientesEnEspera.isEmpty());
}
```