ESTRUCTURA DE DATOS Y ALGORITMOS **22-Junio-2011**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____ Nota:

- Escribe tu **nombre y apellidos** en esta hoja e inmediatamente en todas las suplementarias, incluso las de sucio. El no hacerlo puede suponer tu expulsión
- Puedes utilizar el **lápiz** para tus respuestas. No está permitido el uso de apuntes, notas o libros. No puedes tener un **móvil** encendido, ni utilizar cualquier otro **aparato electrónico**.
- **Todos los alumnos implicados en una copia** de un ejercicio **tendrán una nota final de 0**. El alumno es responsable de velar por su examen. Es decir **tanto el que copia como el que se deja copiar (ya sea de manera activa o pasiva) recibirán el mismo castigo sin que exista atenuante alguno**

1. Ejercicio (2 puntos)

a) Cual de las siguientes afirmaciones es cierta:
   1. Las estructuras de datos almacenan operaciones y las aplican sobre un objeto
   2. **Las estructuras de datos almacenan referencias a objetos**
   3. Las estructuras de datos restringen el uso de las variables
   4. Las estructuras de datos almacenan variables de objetos

b) En un algoritmo de ordenación genérica, hay un método que es la clave del proceso. Escribe la signatura del método. Dónde está definido y/o se implementa?

```
public int compareTo(Object obj)
```
Se define en la interfaz `Comparable` del paquete `java.util` y se implementa en la clase que representa la información que se quiere ordenar.

c) La probabilidad de no encontrar suficiente memoria disponible
   1. **Es mayor para un vector que para una lista**
   2. Es mayor para una lista que para un vector
   3. La probabilidad es la misma para el vector que para la lista
   4. Si la lista es doblemente enlazada 2. es cierto; si es simplemente enlazada 1. es cierto

d) Vamos a suponer que en un mismo programa se han definido dos instancias de la clase iteradora sobre una misma lista enlazada simple. ¿Se pueden producir problemas? En caso afirmativo, explica cómo o cuando puede ocurrir. En caso negativo, explica cual es la característica principal para que no ocurra ningún problema.

Su uso puede ser problemático, exactamente en caso de que si una de las iteradoras elimina justo el nodo de la lista que está referenciado por la variable `current` de la otra iteradora.

e) Una función hash:
   1. **Asigna a una clave una posición en una tabla**
   2. Coloca el valor asociado a una clave dentro de una tabla
   3. Resuelve los conflictos provocados por claves idénticas
   4. Cambia el valor de una clave al de un índice dentro de la tabla

ESTRUCTURA DE DATOS Y ALGORITMOS **22-Junio-2011**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____

2. *Ejercicio (2 puntos)*

```
/**
 * Devuelve True si todas las etiquetas del fichero html están bien formadas
 * @param etiqueta es una etiqueta de tipo HTML
 */
public boolean etiquetasEquilibradas(String fichero){
   Stack etiquetas = new Stack(); //utilizaremos la pila para apilar las etiquetas de
                                  // apertura y desapilaremos la etiqueta en caso de que
                                  // coincida su nombre con la etiqueta de cierre
   boolean existeError = false; // controla si las etiquetas están bien formadas:
                                  // Se produce un error cuando una etiqueta de cierre
                                  // no coincide con la última de apertura.
   Token html = new Token(fichero);
   while (html.hasMoreTokens() && !existeError){
      String palabra = html.getToken();
      if (esEtiqueta(palabra){
        if (esApertura(palabra))
            etiquetas.push(palabra);
        else { //esCierre
            if (etiquetas.isEmpty())
               existeError = true;
            else {
                  String etiqueta = (String) etiquetas.top();
                  if (sonEtiquetasIguales(etiqueta, palabra))
                        etiquetas.pop();
                  else
                        existeError = true;
            }
        }
   }
   return !existeError && etiquetas.isEmpty();
}

/**
 * Devuelve True si la palabra es una etiqueta de apertura o cierre
 * @param palabra es una cadena de caracteres
 */
public boolean esEtiqueta(String palabra){
   return  palabra.charAt(0)=="<" &&
           palabra.charAt(palabra.getLength()-1)==">";
}

/**
 * Devuelve True si la etiqueta es de apertura
 * @param etiqueta es una etiqueta de tipo HTML
 */
public boolean esApertura(String etiqueta){
   return  palabra.charAt(1)!="/";
}

/**
 * Devuelve True si el nombre de las etiquetas de apertura y cierre son el mismo
 * @param abre es una etiqueta de tipo HTML
 * @param cierra es una etiqueta de tipo HTML
 */
public boolean sonEtiquetasIguales(String apertura, String cierre){
   String nombreApertura = apertura.substring(1, apertura.getLength()-2);
   String nombreCierre = cierre.substring(2, cierre.getLength()-3);
   return nombreApertura.equals(nombreCierre);
}
```

3. *Ejercicio (3 puntos)*

a)

```
/**
 * Esta clase representa la información de los pasajeros
 */
class Pasajero{
    private String dni;
    private String nombre;
    private String nacionalidad;

    // los métodos getter
}

/**
 * Esta clase atiende a una fila de pasajeros. Como es normal, el primer pasajero en
 * ser atendido es el primero de la fila y el último que ha llegado se pone al
 * final de la fila. Cómo algunas veces los pasajeros no van a poder ser atendidos,
 * los del final de la fila pueden pasarse a otra fila que tenga menos pasajeros,
 * poniéndose al final.
 */
class Puesto{
     /**
      * como estructura de datos interna seleccionamos una lista doblemente enlazada,
      * para que las operaciones que más a menudo se ejecuten (atender pasajero o
      * equilibrar por el final) sean lo más eficientes posibles. En este caso de
      * tiempo O(1)
      */
     private DoubleLinkedListItr fila;
     /**
      * Devuelve True si la fila del puesto está vacía
      */
     public boolean estaVacio(){...}
     /**
      * Devuelve el número de pasajeros en la fila
      */
     public int numPasajeros(){...}
     /**
      * Devuelve el primer pasajero de la fila
      */
     public Pasajero primeroDeFila(){...}
     /**
      * Pone al pasajero ‘p’ al final de la fila
      */
     public void ponerEnFila(Pasajero p){...}
     /**
      * Atiende al primer pasajero de la fila y quitándolo de ella
      * @return devuelve el pasajero que ha sido atendido
      */
     public Pasajero atiendePasajero(){...}
     /**
      * El último de la fila se pasa al final de la fila del puesto ‘p’
      * @return devuelve el pasajero que se ha cambiado de fila
      */
     public Pasajero pasarAOtroPuesto(Puesto p){...}
}
```

```
/**
 * Esta clase representa la información de la aduana e implementa la funcionalidad
 * requerida.
 */
class Aduana{
     public static final int NUM_PUESTOS = 12;
     /**
      * Como los puestos de atención de la aduana es un número fijo, utilizamos un
      * array
      */
     private Puesto[] puestos = new Puesto[NUM_PUESTOS];
     /**
      * Devuelve True si en la iteración 'it' el puesto 'f' atiende a un nuevo
      * pasajero
      */
     public boolean atiendePasajero(int it, int f){...}
     public void equilibrarFilas(){...}
     public int[] simular(){...}
     ...
}
```

b)

```
/**
 * Equilibra las filas de la aduana, según la estrategia descrita en el enunciado
 */
public void equilibrarFilas(){
   for(int i=0; i<NUM_PUESTOS; i++)
      for(int j=0; j<NUM_PUESTOS; j++){
        if (puestos[i].numPasajeros() > puestos[j].numPasajeros() + 1)
            puestos[i].pasarAOtroPuesto(puestos[j]);
      }
}

class Puesto{
     ...
     public Pasajero pasarAOtroPuesto(Puesto puesto){
        Pasajero p = (Pasajero) fila.removeLast();
        puesto.ponerEnFila(p);
        return p;
      }
     public void ponerEnFila(Pasajero p){
        fila.insertLast(p);
      }
}

class DoubleLinkedListItr{
      private DoubleLinkedList theList;

     /**
      * Inserta un objeto en la parte final de la lista
      * @param obj un objeto
      */
      public void insertLast(Object obj){
            Node nodo = new Node(obj);
            if (theList.isEmpty())
                  theList.top = nodo;
            theList.bottom.next = nodo;
            nodo.previous = theList.bottom;
            theList.bottom = nodo;
      }
```

```
     /**
      * Elimina el último nodo o objeto de la lista
      * @return devuelve el objeto eliminado de la lista
      */
      public Object removeLast(){
            if (theLis.isEmpty())
                  throws new ListaVaciaException();
            Object obj = theList.bottom.elem;
            theList.bottom = theList.bottom.previos;
            if (theList.bottom == null)
                  theList.top = null;
            else
                  theList.bottom.next = null;
            return obj;
      }
```

// Faltaría añadir la definiciones de las clases `DoubleLinkedList` y `Node`.

c)

```
public int[] simular(){
  int[] atendidos = new int[NUM_PUESTOS];
  int pasajeros = 0;
  int it = 1;

  for (int i = 0; i < NUM_PUESTOS;i++){
    atendidos[i]=0;
    pasajeros += puestos[i].numPasajeros();
  }

  while (pasajeros > 0){
    for (int i = 0; I < NUM_PUESTOS;i++){
      if (!puestos[i].estVacio() && atiendePasajero(it, i)){
            atendidos[i]++;
            pasajeros--;
            puestos[i].atiendePasajero();
      }
    }
    if (pasajeros > 0)
      equilibrar();

    it++;
  }

  return atendidos;
}
```

4. Ejercicio (3 puntos)

1. Especificar / Parametrizar

Entrada: A, un árbol binario que representa un árbol binario de números enteros
Salida: La suma de los nodos, cuya altura coincide con su profundidad.

```
public int sumaNodosAlturaProfundidad ();
```

Las definiciones de profundidad y altura vistas en clase son las siguientes:
- La profundidad de un nodo es la longitud del camino desde la raíz hasta el nodo n.
- La altura de un nodo n: es la longitud del camino que va desde el nodo n hasta la hoja más profunda bajo él.

Y si queremos resolver el problema de forma recursiva puede que nos interese más sus definiciones recursivas, que son las siguientes:

- La profundidad del nodo n es: * Si es el nodo es raíz: profundidad = 0 (caso base)
  * En otro caso: profundidad n = 1 + profundidad nodo padre

- La altura de un nodo n: * Si es una hoja (nodo externo): Altura = 0 (caso base)
  *En otro caso: altura de n = 1 + máxima altura de sus hijos

Para resolver el problema de forma recursiva utilizaremos la definición recursiva de árboles binarios, representado mediante la clase `BTNode` e iremos calculando **la altura y la profundidad del nodo raíz** del árbol (subárbol) correspondiente. Por otro lado, para poder calcular la profundidad de un nodo necesitamos saber la profundidad de su nodo padre (parámetro de entrada). En cambio, para la altura, la altura de sus nodos hijos (parámetro de salida).

Analizándolo con un ejemplo, por un lado nos damos cuenta que una vez que en un nodo coincidan su altura y profundidad, es imposible que en alguno de sus subárboles coincidan estas dos propiedades, pero no nos salvamos de recorrerlas porque hay que calcular su altura. Por otro lado, en el caso de que no coincidan las propiedades, puede darse el caso de que se produzca en algún nodo de sus subárboles. Por ello, en el mismo momento que las recorremos para calcular la altura, también aprovecharemos para sumar los nodos de los subárboles (parámetro de salida) cuya altura coincida con su profundidad.

Resumiendo, para resolver el problema de forma recursiva, especificaremos de nuevo el problema/programa aplicando la técnica de *inmersión*, porque necesitamos ampliar los parámetros de entrada y salida.

Entrada: *A*, un árbol binario que representa un árbol binario de números enteros y *un número entero*, que representa la profundidad del nodo padre del nodo raíz de A.
Salida: *dos enteros,* que representan la altura del nodo raíz del árbol A y la suma de los nodos, cuya altura coincide con su profundidad. Para que Java pueda devolver dos valores definiremos la clase `AlturaYSuma`.

```
public class AlturaYSuma {
   int altura;
   int suma;
}

private AlturaYSuma sumaNodosAlturaProfundidad1 (BTNode A, int profPadre)
```

2. Diseño

Casos triviales

Si esVacio (A) → **devolver** new AlturaYSuma(-1, 0);
Si esHoja (A) → altura = 0;
profNodo = profPadre+1
si (profNodo == altura)
suma = A.raiz;
si no
suma =0;
devolver new AlturaYSuma(0, suma);

Caso General

Si noEsVacio(A) y noEsHoja(A) →
profNodo = profPadre+1
altSumIzq = ***sumaNodosAlturaProfundidad1*** (subarbolIzq(A), profNodo);
altSumDch = ***sumaNodosAlturaProfundidad1*** (subarbolDch(A), profNodo);
altura = max(altSumIzq.altura, altSumDch.altura)+1;
si (profNodo == altura)
suma = A.raiz;
si no
suma = altSumIzq.suma + altSumDch.suma;
devolver new AlturaYSuma(altura, suma);

3-Implementación

```
public int sumaNodosAlturaProfundidad1 (){
   SumaYAltura sa = sumaNodosAlturaProfundidad1 (bTree.root, -1);
   System.out.println("La suma es : (" + sa.suma + ")";
}

private SumaYAltura sumaNodosAlturaProfundidad1 (BTNode A, int profPadre) {
   int suma = 0;
   if (A == null)
      return new SumaYAltura(-1, suma);
   else {
      int altura;
      int profNodo = profPadre + 1;
      if ((A.getLeft() == null) && (A.getRight() == null))  {
         altura = 0;
      else {
         SumaYAltura saIzq= sumaNodosAlturaProfundidad1(A.getLeft(),profNodo);
         SumaYAltura saDch= sumaNodosAlturaProfundidad1(A.getRight(),profNodo);
         altura = Math.max(saIzq.altura, saDch.altura) + 1;
         suma = saIzq.suma + saDch.suma;
      }
      if (profNodo == altura)
         suma = ((Integer)A.getContent()).intValue();
   }
   return new SumaYAltura(altura, suma);

}
```