ESTRUCTURA DE DATOS Y ALGORITMOS **6-Septiembre-2010**
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 (3 puntos)

a) Relaciona la lista de descripciones al menos con uno de los nombres de las estructuras de datos que están a la derecha.

1. Restricción de operaciones sobre una estructura de datos	(**4**) Arrays
2. Organización de datos no lineal	(**3**) Vectores
3. En su definición el uso de la memoria es estática, pero puede variar en la ejecución de un programa	(**1**) Pilas
4. Uso ineficiente de la memoria	(**2, 5**) Árboles
5. La inserción de un nuevo dato requiere analizar varias casos	(**5**) Lista

b) En qué estructura de datos se utiliza el término *colisión*? A qué se refiere exactamente este término?

Se utiliza en las tablas hsh cuando al insertar un nuevo elemento en la tabla, la función hash devuelve la misma posición que otro elemento ya insertado y con diferente clave.

En las **tablas hash**. **Se utiliza para indicar que vamos a tener problemas en las operaciones sobre la estructura de datos con las claves de distintos datos que son sinónimos**. Son sinónimos porque después de aplicar la función hash a distintas claves se obtiene el mismo valor. Este valor se utiliza como índice en el array interno de la tabla hash, p. ej para insertar el dato, y puede ocurrir que esa posición esté ya ocupada por otro dato. Por lo tanto, se ha producido colisión entre las dos claves: el almacenado y el que se va a insertar.

c) La estructura de datos abstracta Pila se conoce también como una estructura **LIFO**. Explica el porqué.

LIFO es el acrónimo de **Last-In First-Out**, lo que significa que el último dato en entrar es el primero en salir y eso es lo que realizan las operaciones principales push (insertar) y pop (eliminar) de la pila, respectivamente.

d) La solución de un problema concreto ejecuta, con bastante frecuencia, las dos siguientes operaciones sobre un conjunto de datos: 1) buscar un dato por contenido y 2) insertar o eliminar más datos después o antes del dato encontrado en el 1). ¿Qué estructura de datos sería la más adecuada? Argumenta tu decisión.

Una **lista doblemente enlazada** sería la más adecuada, porque al no utilizar una estructura estática en memoria para la representación de los datos, las operaciones comentadas no requieren desplazamiento de los datos anteriores y además hace un uso justo de la memoria requerida. Cualquier otra implementación de las listas enlazadas (simples, con doble terminación o circulares), no serían validas porque no se podrían manipular los datos anteriores al dato encontrado. Por lo menos, en tiempo constante. También podría ser valido un **árbol** con los mismos argumentos.

e) 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 restringen el uso de las variables
   3. Las estructuras de datos almacenan variables de objetos
   4. **Las estructuras de datos almacenan referencias a objetos**

f) Las principales diferencias entre las distintas estructuras son:
   1. La forma de organizar las operaciones y las restricciones que imponen sobre los datos
   2. **La forma de organizar los datos y las restricciones que imponen sobre las operaciones**
   3. La forma en que aplican la abstracción
   4. En principio, no existe ninguna diferencia sustancial, depende del problema planteado

g) Las tablas hash son utilizadas porque:
   1. Se deben de recorrer los elementos en orden
   2. No se conoce el número de elementos a priori
   3. Se quiere utilizar un mismo índice para varios elementos
   4. **Ninguna de las anteriores**

h) Un vector tiene la siguiente propiedad:
   1. Inserciones y borrados eficientes en medio de la estructura
   2. Inserciones y borrados eficientes en ambos extremos de la estructura
   3. **Inserciones y borrados eficientes al final de la estructura**
   4. Ninguna de las anteriores

ESTRUCTURA DE DATOS Y ALGORITMOS **6-Septiembre-2010**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____

- 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**

2. *Ejercicio (2,5 puntos)*

a) (0,75 puntos)

Por un lado necesitamos una estructura de datos que mantenga los datos ordenados, p.ej. VectorOrdenado o ArrayOrdenado, utilizando las interfaces Comparable (sólo un criterio) o Comparator (varios criterios). En nuestro caso, utilizaremos el Comparator porque se quiere ordenar por varios atributos. Exactamente, utilizaremos un Comparator por cada atributo.

```
public class VerBolsa{

   private VectorOrdenado listaValores; // el vector está siempre ordenado según el último
    // criterio del usuario, el cual es almacenado en las variables ultimoComparador y ultimoOrden
   private Hashtable comparadores; // se utiliza para almacenar varios comparadores distintos,
                                   // uno por cada atributo de la clase Valor
   private Comparator ultimoComparador;   // define el último comparador seleccionado por el
                                          // usuario
   private boolean ultimoOrden; //True significa "ascendente" y False, "descendente"

   public VerBolsa(){
      listaValores = new VectorOrdenado();
      comparadores = new Hashtable();
      Comparator comparador = new ValorPorNombre("porNombre");
      comparadores.put(comparador.getKey(), comparador.getValue();
      Comparator comparador = new ValorPorPrecio("porPrecio");
      comparadores.put(comparador.getKey(), comparador.getValue();
      Comparator comparador = new ValorPorPorcentaje("porPorcentaje");
      comparadores.put(comparador.getKey(), comparador.getValue();
      Comparator comparador = new ValorPorVolumen("porVolumen");
      comparadores.put(comparador.getKey(), comparador.getValue();
      ultimoComparador = comparadores.get("porNombre");
      ultimoOrden = true;
   }
}

public abstract class ValorPorAtributo implements Entry{
   private String nombreAtributo;
   public ValorPorAtributo(String nombre){
         nombreAtributo = nombre;
   }
   public Object getkey(){
         return nombreAtributo;
   }
   public Object getValue(){
         return this;
   }
}
```

```
public class ValorPorNombre extends ValorPorAtributo implements Comparator{
   public int compare(Object obj1, Object obj2){
         Valor v1 = (Valor) obj1;
         Valor v2 = (Valor) obj2;
         return v1.getNombre().compareTo(v.2.getNombre();
   }
}
```

Implementaremos 3 clases más, las cuales extenderán la clase `ValorPorAtributo` e implementarán la interfaz `Comparator` de la misma forma que la clase anterior, cuyos nombres son: `ValorPorPrecio, ValorPorPorcentaje` y `ValorPorVolumen`

b) (1,5 punto)

```
public class VerBolsa{
    /**
     * Carga desde un fichero de texto los datos de los valores según el cierre del día anterior
     */
       public void cargarDatos(String fichero){
          Token datos = new Token(fichero);
          while datos.hasMoreTokens(){
            Valor valor = datos.getToken();
            insertarValorOrdenado(valor);
          }
       }
    /**
     * Actualiza los datos de los valores según el fichero recibido con los nuevos datos
     */
       public void actualizar(String fichero) {
          Token datos = new Token(fichero);
          while datos.hasMoreTokens(){
            Valor valor = datos.getToken();
            listaValores.remove(valor);
            insertarValorOrdenado(valor);
          }
       }
    /**
     * Inserta el valor en la lista según el criterio de ordenación
     */
       public void insertarValorOrdenado(Valor valor){
         listaValores.insertSorted(valor, ultimoComparador, ultimoOrden);
       }
    /**
     * Cambia el criterio de ordenación, ordena y visualiza la lista
     */
       public void cambiarCriterioOrdenacion(String nombre, boolean orden){
         ultimoComparador = (Comparator) comparadores.get(nombre);
         ultimoOrden = orden;
         insertionSort(listaValores, ultimoComparador, ultimoOrden);
         visualizar();
       }
    /**
        * Visualiza la lista de valores según el criterio de ordenación
        */
       public void visualizar(){
         Enumeration valores = listaValores.elements();
         while valores.hasMoreElements(){
           Valor valor = (Valor) valores.next();
           System.out.println(valor.toString());
         }
       }
}
```

```
public class VectorOrdenado{
   private Vector vector;
   private Comparator comp;
   public void insertSorted(Object obj, Comparator comp, boolean order){
         int pos = 0;
         boolean esMenor = false; //describe que obj es menor que cualquier objeto
                                  // del vector analizado hasta la posición pos-1
         while ((pos < vector.length) && !esMenor){
               if (comp.compare(obj, vector.elementAt(pos)) < 0)
                     esMenor = True;
               pos++;
         }
         If (esMenor)
               vector.insertElementAt(pos, obj);
         esle
               vector.addElement(obj)
   }
}

public void insertionSort(Vector v, Comparator comp, boolean order){
    // se podía haber seleccionado cualquier otro algoritmo de ordenación

}
```

c) (0,25 puntos) Calcula cual es la complejidad, en notación *O*, de los métodos implementados en b).

La complejidad de los métodos implementados en el apartado a) los evaluaremos respecto al número de datos que vamos a gestionar, que en este caso corresponde exactamente con el número de valores en Bolsa. Llamemos *n* a dicho número.

Los métodos cargarDatos y cambiarCriterioDeOrdenación son de orden cuadrático $O(n^2)$. El método actualizar de $O(m*n)$, siendo *m* el número de valores que se actualizan en cada momento. Finalmente, decir que los otros métodos restantes son de orden lineal $O(n)$.

Empezando por estos últimos, la complejidad del método visualizar es lineal $O(n)$, porque simplemente realizamos un recorrido sobre la estructura de datos. En el caso del método insertarValorOrdenado es lineal $O(n)$ porque depende directamente del método insertSorted, cuya complejidad es lineal porque en el peor de los casos recorre toda la estructura de datos para insertar el dato ordenado al final.

Siguiendo con el método actualizar, las operaciones principales que se ejecutan son el remove y el insertarValorOrdenado, cuyas complejidades son lineales por el hecho de que en el peor de los casos hay que realizar un recorrido sobre la estructura de datos. Dichas operaciones se repiten según el número de valores que hay que actualizar en cada momento, al cual le hemos llamado *m*. Por ello, la complejidad del método actualizar es $O(m*n)$.

Para terminar, decir que la complejidad de cambiarCriterioDeOrdenación depende principalmente del algoritmo de ordenación utilizado. En nuestro caso es insertionSort, cuya complejidad es de $O(n^2)$. En el caso de cargarDatos, puede chocar que sea de $O(n^2)$ porque se está construyendo la estructura de datos desde cero y se esperaba que fuera lineal. Eso sería así si los datos del fichero se introducen tal cual en la estructura de datos. Pero no es el caso, porque se quiere construir una estructura que tenga los datos ordenados.

Nota: Se valorarán los aspectos de reutilización.

3. *Ejercicio (1,5 punto)*

Los casos que vamos a tener en cuenta son los siguientes y en cada caso, indicaremos cual es el estado de las referencias "lento" y "rápido" justo antes de que se produzca la división: (falta añadir imágenes)

a) La lista es vacía

b) El número de elementos que tiene la lista es impar. Analizamos las listas de 1 y 3 elementos.

c) El número de elementos que tiene la lista es par. Analizamos las listas de 2 y 4 elementos.

```
public class LinkedListItr {
   ...
  public FrenteTrasera divideFrenteTrasera(){
      FrenteTrasera resultado = new FrenteTrasera(); //crea dos listas vacías
      if (!theList.isEmpty()){
            Node lento = theList.top;
            Node rapido = theList.top.next;
            // mientras no ésta al final (caso b)) o no está en el último nodo (caso c))
            while ((rapido != null) && (rapido.next != null)){
                  lento = lento.next;
                  rapido = rapido.next.next
            }
            // se divide la lista, dejando el nodo 'lento' como último nodo de la sublista frente.
            // La trasera empieza en el siguiente nodo que referencia el nodo 'lento'.
            frente.top = theList.top;
            trasera.top = slow.next;
            slow.next = null;
      }
      return resultado;
  }
}
```

5. ***Ejercicio (3 puntos)***

Tenemos dos opciones para resolver dicho problema: procesar los capítulos en pre-order o post-order:

Procesando los capítulos en post-order

1. Especificar / Parametrizar

Entrada: A, un árbol binario que representa un cuento dinámico completo.
Salida: un vector, el cual representa un conjunto de cuentos individuales en el árbol A

```
public Vector imprimirCuentosIndividuales();
```

Para resolver el problema utilizaremos la definición recursiva de árboles binarios, representado mediante la clase BTNode.

```
private Vector imprimirCuentosIndividuales (BTNode A)
```

2. Diseño

Casos triviales

Si esVacio (A) → **devolver** new Vector()
Si esHoja (A) → v = new Vector(); v.addElement(raiz(A)); **devolver** v

Caso General

Si noEsVacio(A) y noEsHoja(A) →
vectorIzq = ***imprimirCuentosIndividuales***(subarbolIzq(A), cuento, true);
anadirInicioACuentos(raiz(A)+"-->si-->", vectorIzq);
vectorDch = ***imprimirCuentosIndividuales***(subarbolDch(A), cuento, false);
anadirInicioACuentos(raiz(A)+"-->no-->", vectorDch);
añadirTodosEnVector1(vectorIzq, vectorDch)
devolver vectorIzq;
fin si

3-Implementación

```
public Vector imprimirCuentosIndividuales(){
   Vector cuentos = imprimirCuentosIndividuales(bTree.root);
   return cuentos;
}

private Vector imprimirCuentosIndividuales (BTNode A) {
   if (A == null) return new Vector();
   else {
         String titulo = ((Capitulo) A.getContent()).getTitulo();
         if (A.left == null && A.right == null){
               Vector v = new Vector();
               v.addElement(titulo);
               return v;
         }
         else {
               Vector vIzq = imprimirCuentosIndividuales(A.left, cuento, true);
               anadirInicioACuentos (titulo  & "-->si-->", vectorIzq);
               Vector vDch = imprimirCuentosIndividuales(A.right, cuuento, false);
               anadirInicioACuentos(titulo  & "-->no-->", vectorIzq);
               vIzq.addAll(vDch);
               return vIzq;
         }
   }
}
```

```
private void anadirInicioACuentos (String inicio, Vector cuentos){
   for (int i = 0;i < cuentos.size();i++){
         String cuento = (String) cuentos.elementAt(i);
         cuento = inicio + cuento;
         cuentos.setElementAt(i, cuento);
   }
}
```

ESTRUCTURA DE DATOS Y ALGORITMOS **6-Septiembre-2010**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

Procesando los capítulos en pre-order

1. Especificar / Parametrizar

Entrada: A, un árbol binario que representa un cuento dinámico completo.
Salida: un vector, el cual representa un conjunto de cuentos individuales en el árbol A

```
public Vector imprimirCuentosIndividuales();
```

Pars resolver el problema de forma recursiva utilizaremos la definición recursiva del árbol binario, representado mediante la clase BTNode.

A la hora componer un cuento individual, nos damos cuenta 1) que es equivalente a crear o recorrer un camino de nodos (desde la raíz hasta una hoja) en un árbol binario. Por ello, a la hora de procesar un nodo del árbol necesitamos saber el camino recorrido hasta el momento, es decir, desde la raíz hasta su nodo padre; y 2) antes de añadir un capítulo más al cuento individual, necesitamos saber la respuesta que hay que seleccionar en el capitulo anterior para poder llegar al capitulo que se quiere añadir. Por lo tanto, este problema requiere del mecanismo de INMERSION.

Entrada: - A, un árbol binario que representa parte del cuento dinámico;
- una cadena de caracteres, que representa los capítulos iniciales de un cuento individual, es decir, los capítulos necesarios que hay que leer para llegar desde el inicio del cuento dinámico inicial hasta el primer capitulo de A.
- un booleano, que indica la respuesta de la pregunta del capitulo anterior al primer capitulo de A.

Salida: un vector, el cual representa un conjunto de cuentos individuales.

```
private Vector imprimirCuentosIndividuales (BTNode A, String cuento,
                                            boolean respuesta);
```

2. Diseño

Casos triviales

Si esVacio (A) → **devolver** new Vector();
Si esHoja (A) → cuento = cuento + respuesta + raiz(A);
devolver new Vector().addElement(cuento)

Caso General

Si noEsVacio(A) y noEsHoja(A) →
Si esVacio(cuento) **entonces**
cuento = raiz(A);
si no
cuento = cuento + respuesta + raiz(A);
fin si
vectorIzq = ***imprimirCuentosIndividuales***(subarbolIzq(A), cuento, true);
vectorDch = ***imprimirCuentosIndividuales***(subarbolDch(A), cuento, false);
añadirTodosEnVector1(vectorIzq, vectorDch)
devolver vectorIzq;
fin si

Nota: para entender mejor la solución se ha utilizado directamente el valor del parámetro respuesta, pero en la implementación habría que sustituirlo por su equivalente cadena de caracteres.

3-Implementación

```
public Vector imprimirCuentosIndividuales(){
   Vector cuentos = imprimirCuentosIndividuales(bTree.root, “”, false);
   return cuentos;
}

private Vector imprimirCuentosIndividuales (BTNode A, String cuento,
                                            boolean respuesta){
   if (A == null) return new Vector();
   else {
         String titulo = ((Capitulo) A.getContent()).getTitulo();
         if (cuento.equals(“”))
               cuento = titulo;
         else if (respuesta)
               cuento = cuento & “-->si-->” & titulo;
         else
               cuento = cuento & “-->no-->” & titulo;

         if (A.left == null && A.right == null){
               Vector v = new Vector();
               v.addElement(cuento);
               return v;
         }
         else {
               Vector vIzq = imprimirCuentosIndividuales(A.left, cuento, true);
               Vector vDch = imprimirCuentosIndividuales(A.right, cuuento, false);
               vIzq.addAll(vDch);
               return vIzq;
         }
   }
}
```