ESTRUCTURA DE DATOS Y ALGORITMOS **29-Enero-2009**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____ Nota:

NOTA FINAL: Nota Practica (1 punto) + Nota Examen (9 punto)

- Es indispensable aprobar el examen (4,5 puntos) para aprobar la asignatura (5 puntos)
- La práctica es opcional
- Duración: 3 horas
- No está permitido el uso de apuntes, libros o móviles
- **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**
- Publicación de las notas: 15-2-2009
- Revisión del examen: 16-2-2009, 15:00 - 18:00

1. Ejercicio (2 puntos)

a) Una función hash:
   A. Asigna a una clave una posición en una tabla
   B. Coloca el valor asociado a una clave dentro de una tabla
   C. Resuelve los conflictos provocados por claves idénticas
   D. Cambia el valor de una clave al de un índice dentro de la tabla

b) Se han insertado un conjunto de datos en una estructura de datos para su procesamiento. Debido a la característica de la estructura de datos, después de la inserción los datos están siendo procesados en orden inverso al orden de inserción. ¿Qué estructura de datos se está utilizando?

Pila

c) Si consideramos básica la operación ***vacíar*** sobre pilas que vacía completamente una pila y la implementamos en Java:
   A. La operación implementada con arrays requiere del recorrido de todo el array y mediante listas enlazadas de solo una operación.
   B. La operación implementada con arrays y mediante listas enlazadas requieren una sola operación.
   C. La operación implementada con arrays y mediante listas enlazadas requiere recorrer todo la estructura.
   D. No es válida ninguna de las anteriores.

d) Explica cuando, como y donde se debe implementar el método *"equals"*.

El método "equals" se suele utilizar para la comparación de dos objetos y se implementa en la clase del objeto que se quiere comparar. Para ello, recibe por parámetro el objeto a comparar con el objeto actual y se comparan aquellos atributos que son suficientes para decidir si un objeto es igual a otro.

Dicho esto, en los programas complejos dónde normalmente se tienen que gestionar un conjunto de datos o información, es decir objetos, estas se organizan o almacenan en estructuras de datos. A menudo se realizan operaciones de búsqueda y/o eliminación en las estructuras de datos, los cuales requieren o delegan la comparación de dos objetos en el método "equals". Por tanto, vuelvo a comentar que este método se implementaría en la clase que represente la información del problema en cuestión.

ESTRUCTURA DE DATOS Y ALGORITMOS **29-Enero-2009**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____ Nota:

NOTA FINAL: Nota Practica (1 punto) + Nota Examen (9 punto)

- Es indispensable aprobar el examen (4,5 puntos) para aprobar la asignatura (5 puntos)
- La práctica es opcional
- Duración: 3 horas
- No está permitido el uso de apuntes, libros o móviles
- **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**
- Publicación de las notas: 15-2-2009
- Revisión del examen: 16-2-2009, 15:00 - 18:00

1. Ejercicio (2 puntos)

a) Si consideramos básica la operación ***vacíar*** sobre pilas que vacía completamente una pila y la implementamos en Java:
   A. La operación implementada con arrays requiere del recorrido de todo el array y mediante listas enlazadas de solo una operación.
   B. La operación implementada con arrays y mediante listas enlazadas requieren una sola operación.
   C. La operación implementada con arrays y mediante listas enlazadas requiere recorrer todo la estructura.
   D. No es válida ninguna de las anteriores.

b) Se han insertado un conjunto de datos en una estructura de datos para su procesamiento. Debido a la característica de la estructura de datos, después de la inserción los datos están siendo procesados en orden inverso al orden de inserción. ¿Qué estructura de datos se está utilizando?

Pila

c) Una función hash:
   A. Asigna a una clave una posición en una tabla
   B. Coloca el valor asociado a una clave dentro de una tabla
   C. Resuelve los conflictos provocados por claves idénticas
   D. Cambia el valor de una clave al de un índice dentro de la tabla

d) Explica cuando, como y donde se debe implementar el método *"equals"*.

El método "equals" se suele utilizar para la comparación de dos objetos y se implementa en la clase del objeto que se quiere comparar. Para ello, recibe por parámetro el objeto a comparar con el objeto actual y se comparan aquellos atributos que son suficientes para decidir si un objeto es igual a otro.

Dicho esto, en los programas complejos dónde normalmente se tienen que gestionar un conjunto de datos o información, es decir objetos, estas se organizan o almacenan en estructuras de datos. A menudo se realizan operaciones de búsqueda y/o eliminación en las estructuras de datos, los cuales requieren o delegan la comparación de dos objetos en el método "equals". Por tanto, vuelvo a comentar que este método se implementaría en la clase que represente la información del problema en cuestión.

ESTRUCTURA DE DATOS Y ALGORITMOS **29-Enero-2009**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

Nombre y apellidos: _____

NOTA FINAL: Nota Practica (1 punto) + Nota Examen (9 punto)

- Es indispensable aprobar el examen (4,5 puntos) para aprobar la asignatura (5 puntos)
- La práctica es opcional
- Duración: 3 horas
- No está permitido el uso de apuntes, libros o móviles
- **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**
- Publicación de las notas: 15-2-2009
- Revisión del examen: 16-2-2009, 15:00 - 18:00

2. *Ejercicio (3 puntos)*

Dada la siguiente especificación de las clases BTNode, BinTree y BinTreeItr:

```
public class BinTreeItr{
      BTNode current;
      BinTree bTree;
}

public class BinTree{
      BTNode root;
}
```

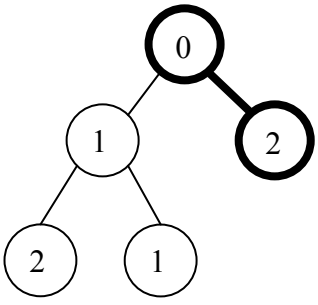
```
public class BTNode{
      Object content;
      BTNode left;
      BTNode right;
}
```

Nota: Todo método necesario tendrá que ser implementado.

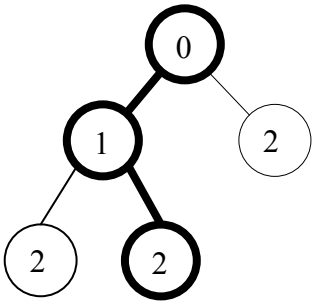
Especificar, diseñar e implementar un método recursivo que dado un árbol binario de enteros, calcule y devuelva cual es la longitud del camino que mayor coste supone recorrerla. En caso de que haya más de un camino con el mismo coste, entonces devolver el de menor longitud. Un **camino** es la secuencia de arcos que une dos nodos de un árbol ~~una secuencia de nodos en la que cada nodo es adyacente al siguiente~~. Mientras, la **longitud** de un camino entre dos nodos es el número de arcos presentes entre ambs nodos. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el nodo raíz y su contenido (el entero) representa el coste de visitarlo. Además, visitar los **nodos derechos cuesta el doble** que visitar los izquierdos, mientras el coste del nodo raíz siempre es 0.

```
int longitudCaminoCosteMaximo();
```

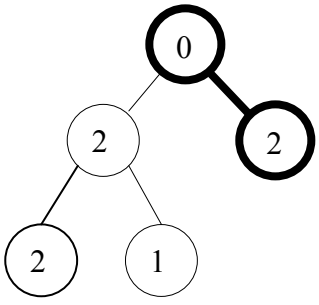
Ejemplos:



Longitud = 1



Longitud = 2



Longitud = 1

ESTRUCTURA DE DATOS Y ALGORITMOS **29-Enero-2009**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

1. Especificar / Parametrizar

Entrada: A, un arbol binario.
Salida: un entero, que representa la longitud del camino de mayor coste

```
public int longitudCaminoCosteMaximo();
```

A la hora de tratar un nodo del árbol y poder calcular la longitud del camino de mayor coste, debemos de conocer tres cosas: 1) si el nodo actual es izquierdo o derecho, para poder calcular el coste total que supone visitar dicho nodo; 2) el coste máximo de recorrer uno de los caminos de los subárboles izquierdo y/o derecho; y 3) la longitud minima de los caminos de los subárboles izquierdo y/o derecho cuyo recorrido suponga el coste máximo → INMERSION

Entrada: A, un arbol binario y esIzquierdo, un booleano que representa si la raíz de A es izquierdo o derecho.
Salida: dos enteros. El primero representa el coste máximo de recorrer uno de los caminos del árbol A y el segundo representa la longitud minima de los caminos cuyo recorrido suponga el coste máximo.

Se necesita una clase Contexto para poder devolver dos valores:

```
public class Contexto {
    int coste;
    int longitud;
}
```

```
private Contexto longitudCaminoCosteMaximo1(BTNode A,boolean esDerecho);
```

2. Diseño

Casos triviales

Si esVacio (A) → devolver new Contexto(0,0)
Si esHoja (A) y no(esDerecho) → devolver new Contexto(raiz(A), 0)
Si esHoja (A) y esDerecho → devolver new Contexto(2*raiz(A), 0)

Caso General

```
Si No Hoja (A) →
   costeA = (esDerecho? 2 : 1) * raiz(A);
   contextoIzq = longitudCaminoCosteMaximo1 (subarbolIzq(A), true);
   contextoDch = longitudCaminoCosteMaximo1 (subarbolDch(A), false);
   Si (contextoIzq. coste > contextoDch.coste) entonces
      coste = contextoIzq.coste + costeA;
      longitud = contextoIzq.longitud + 1;
   si no (contextoIzq. coste < contextoDch.coste) entonces
      coste = contextoDch. coste + costeA;
      longitud = contextoDch.longitud + 1;
   si no
      coste = contextoDch. coste + costeA;
      longitud = minimo(contextoIzq.longitud, contextoDch.longitud) + 1;
 fin si
 contexto = new Contexto(coste, longitud);
 devolver contexto;
```

3-Implementación

```
public int longitudCaminoCosteMaximo() {
      Contexto contexto = longitudCaminoCosteMaximo1 (bTree.root, false);
      return contexto.longitud;
}

private Contexto longitudCaminoCosteMaximo1(BTNode A, boolean esIzquierdo) {
      int coste, longitud;

      if (A==null)
        return new Contexto(0, 0);
      else if ((A.left==null) && (A.right==null) ) {
         coste = (esIzquierdo? 2 : 1) * A.content;
         return new Contexto(coste, 0);
      }
      else {
        int costeA = (esIzquierdo? 2 : 1) * ((Integer)A.content).intValue();
        Contexto contextoIzq = longitudCaminoCosteMaximo1(A.left, true);
        Contexto contextoDch = longitudCaminoCosteMaximo1(A.right, false);
        if (contextoIzq.coste > contextoDch.coste){
          coste = contextoIzq.coste + costeA;
          longitud = contextoIzq.longitud + 1;
        }
        else if (contextoIzq.coste < contextoDch.coste){
          coste = contextoDch.coste + costeA;
          longitud = contextoDch.longitud + 1;
        }
        else { //el coste maximo es el mismo
          coste = contextoDch.coste + costeA;
          longitud = Math.min(contextoIzq.longitud,contextoDch.longitud)+1;
        }
        Contexto contexto = new Contexto(coste, longitud);
        return contexto;
      }
}

public class Contexto {
      int coste, longitud;
      public Contexto(int coste, int longitud) {
        this. altura = altura;
        this.es_lleno = es_lleno;
      }
}
```

3. *Ejercicio (4 puntos)*

Desarrollar un sistema informático que modele el comportamiento de un mercado de trabajo simplificado, donde las personas pueden ser contratadas y despedidas por empresas de forma muy fácil y a menudo. Una persona puede estar trabajando en una empresa o no, y una empresa, como es lógico, tiene contratado a varias personas. Comentar que todas las personas tienen distinto dni, lo mismo que, todas las empresas tienen distinto nif.

Para ello suponemos disponibles la implementación de las siguientes clases, todas ellas equipadas con operaciones de acceso, igualdad (*equals*) y orden (implementan la interfaz *Comparable*, es decir, el método *compareTo*):

- *Persona* que sirve para almacenar y gestionar la información sobre una persona (dni, nombre, dirección, edad)
- *Empresa* que sirve para almacenar y gestionar la información sobre una empresa (nif, nombre, dirección).

Se pide:

a) Diseña e implementa la estructura de la clase *Mercado*, cuya misión es gestionar la información sobre el mercado de trabajo, teniendo en cuenta as características del problema y las estructuras de datos vistas en clase. También implementa las clases necesarias para representar las relaciones entre la información a gestionar. **Nota**: no implementes ni modifiques las clases Persona y Empresa.

b) Implementa las siguientes operaciones públicas de la clase *Mercado*:

- ***Constructor***: Genera un mercado vacío, sin ninguna información.
- ***contrata***: Altera un mercado, efectuando la contratación de cierta persona como empleado de cierta empresa.
- ***despide***: Altera un mercado, efectuando el despido de cierta persona que era antes empleado de cierta empresa.
- ***empleados***: Consulta los empleados de una empresa, devolviendo como resultado un array ordenado de personas.
- ***esEmpleado***: Averigua si es cierto o no que una persona dada es empleado de una empresa dada.

Nota: No es necesario implementar los métodos típicos de cualquier estructura de datos que hayamos visto en clase, siempre que se utilicen los nombres habituales.

a)

Por un lado voy a necesitar dos nuevas clases (Empleado, EmpresaEmpleados) para representar las relaciones entre objetos Empresa y Persona. La primera clase (Empleado) representará que una persona es empleada de una determinada empresa. LA segunda clase (EmpresaEmpleados) representará cuales son los empleados (objetos de tipo Persona) de una determinada empresa.

```
Public class Empleado implements Entry{
   private Persona empleado;
   private Empresa empresa;
}

Public class EmpresaEmpleados implements Entry{
   private Empresa empresa;
   private Hashtable empleados;  // Se van a almacenar objetos de tipo Persona, el cual requiere implementar la interfaz Entry
}
```

Por otro lado, estas relaciones son básicamente la información que va a gestionar la clase Mercado. Para la organización de dicha información, utilizaremos dos tablas hash (empresas y empleados) porque las contrataciones y despidos que se van a producir en un mercado van a ser frecuentes y esto requiere que las operaciones de la clase Mercado sean lo más eficientes posibles.

```
Public class Mercado{
   Private Hashtable empresas;   // se van a almacenar objetos de tipo EmpresaEmpleados y
                                 // en ella se requiere implementar la interfaz Entry
   Private Hashtable empleados;  //se van a almacenar objetos de tipo Empleado y
                                 // en ella se requiere implementar la interfaz Entry
}
```

b)

```
   /**
      * PRE: la persona p no es empleada de la empresa e
      */
   public void contrata(Persona persona, Empresa empresa) {  //implementado en la clase Mercado
      Empleado empleado = new Empleado(persona, empresa);
      empleados.put(empleado.getKey(), empleado.getValue());
      EmpresaEmpleados empresaEmpleados = null;
      //verificar si la empresa existe
      If (empresas.exist(empresa.getKey()))
         empresaEmpleados = (EmpresaEmpleados) empresas.get(empresa.getKey());
      else
         empresaEmpleados = new EmpresaEmpleados(empresa);
       empresaEmpleados.contrata(persona);
   }

  public void contrata(Persona persona){  // implementado en la clase EmpresaEmpleados
      empleados.put(persona.getKey(), persona.getValue());
   }
```

```
   /**
      * PRE: la persona p es empleada de alguna empresa
      */
   public void despide(Persona persona){ //implementado en la clase Mercado
      Empleado empleado = (Empleado) empleados.get(persona.getKey());
      EmpresaEmpleados empresaEmpleados = (EmpresaEmpleados) empresas.get(empleado.getEmpresa().getKey());
      empresaEmpleados.despide(persona);
      empleados.remove(persona.getKey());
   }

  public void despide(Persona persona){ // implementado en la clase EmpresaEmpleados
      empleados.remove(persona.getKey());
   }
```

```
public Persona[] empleados(Empresa empresa){
   //verificar si la empresa existe
   If (empresas.exist(empresa)) {
      empresaEmpleados = (EmpresaEmpleados) empresas.get(empresa);
      Persona[] personas = empresaEmpleados.getArrayEmpleados();  //no están ordenados
      selectionSort(personas)  // usar cualquier otro algoritmo de ordenación es valido
      return personas;
   }
   else
      return new Persona[0];
}

 public Persona[] getArrayEmpleados (){  //implementado en la clase EmpresaEmpleados
    Persona[] personas = new Persona[empleados.size()];
    Enumeration valores = empleados.getValues();
    int pos = 0;
    while (valores.hasMoreElements()) {
        Personas[pos] = valores.nextElement();
        pos++;
    }
    return personas;
}

public void selectionSort(Comparable[] datos)      //implementar un algoritmo de ordenación genérico
```

```
public boolean esEmpleado(Persona persona, Empresa empresa){
   if (empleados.exist(persona)){
      Empleado empleado = (Empleado) empleados.get(persona.getKey());
      return empleado.getEmpresa().equals(empresa);
   }
   else false;
}
```