ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN
ESTRUCTURAS DE DATOS
SEGUNDA EVALUACIÓN - I TÉRMINO 2016

Nombre: _____ ***Matrícula:*** _____

TEMA 1 (12 puntos)

Considere las siguientes estructuras:

HashMap	Grafo Dirigido sin ciclos
Grafo No Dirigido	Árbol Binario
Grafo Dirigido con ciclos	Árbol Binario de Búsqueda

Seleccione y **justifique** la estructura de datos más apropiada para cada uno de los siguientes problemas:

a) Un mapa del tráfico del Ecuador que muestre las ciudades principales, carreteras junto con la dirección de la misma (un sentido o doble sentido), y tiempos entre ciudades.

b) Un manual de instrucciones para construir una computadora donde algunas veces es necesario hacer dos o más pasos antes de hacer un paso en específico.

c) Una lista de palabras válidas del juego del Scrabble. Es necesario que podamos encontrar las palabras hechas por los usuarios de forma rápida y que en verdad existan en la lista.

d) La representación de herencia en Java, donde una clase sólo puede heredar de una clase.

TEMA 2 (8 puntos)

Dado los siguientes 3 arreglos:

	0	1	2	3	4	5	6	7	8	9
A =	99	64	6	54	32	28	42	19	7	

	0	1	2	3	4	5	6	7	8	9
B =	99	64	42	54	32	28	6	19	7	

	0	1	2	3	4	5	6	7	8	9
C =	99	64	42	19	32	28	6	54	7	

a) Determine cuál de los arreglos representa un heap?

b) Del literal a, cuál es el arreglo resultante después de aplicar la operación **encolar(65)**?

c) A partir del literal b, cuál es el arreglo resultante después de aplicar la operación **desencolar()**?

Tema 3 (15 puntos)
Una empresa de streaming media ha decidido incorporar un servicio de envío online de letras de canciones, para lo cual utiliza compresión de texto basada en Códigos de Huffman, donde cada ***palabra*** recibe un código binario. Cuál sería una posible codificación para la siguiente estrofa:

You say yes I say no
You say stop and I say go go go oh no
You say goodbye and I say hello
Hello hello

a) Grafique el Árbol de Huffman correspondiente

b) Escriba los códigos de Huffman de cada palabra

*Nota: Existen 11 palabras diferentes en la estrofa

Tema 4 (15 puntos)
Considere un HashMap<String,LinkedList> que tiene como clave un usuario de Twitter y como valor una lista de mensajes que le pertenecen al usuario. Los mensajes pueden contener menciones a otros usuarios de Twitter.

```
{
    "@user1" : ["Saludos @user2 y @user3" , "Respondeme @user2"] ,
    "@user2" : ["Hola @user3 y @user4" , "Saludos @user4"] ,
    "@user3" : ["Hoy me siento genial" , "Es un hermoso día"] ,
    ...
}
```

A usted se le solicita implementar la función **mencionesPorUsuario**, la función recibe el mapa con los mensajes de todos los usuarios y retorna un HashMap<String,LinkedList> donde la clave es un usuario de Twitter y el valor es una lista con los usuarios que lo han mencionado.

```
{
    "@user2":["@user1"] ,
    "@user3":["@user1" , "@user2"] ,
    "@user4":["@user2"] ,
    ...
}
```

```
public static HashMap<String,LinkedList> mencionesPorUsuario(HashMap<String,LinkedList> tw)
```

Tema 5 (25 puntos)

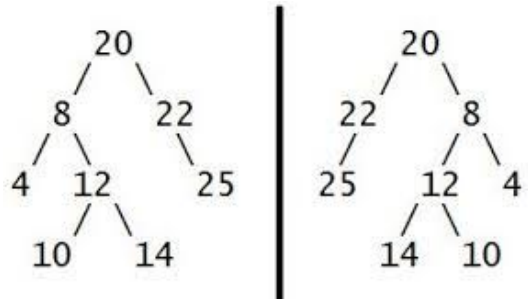
Considerando las siguientes definiciones de TDAs necesarias para representar un Árbol Binario (AB):

```
public class Nodo
{
        private Nodo izquierdo, derecho;
        private Object contenido;
        ….
}
public class AB
{
        private Nodo raiz;
        ….
}
```

A usted se le solicita implementar las siguientes funciones en el TDA Árbol Binario (AB):

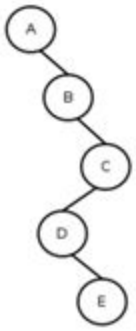
1) **(15 puntos)** El método **espejo** que retorna un nuevo árbol binario con el espejo del árbol. Un árbol espejo es aquel que cada nodo tiene cambiado su hijo izquierdo por el derecho y viceversa.

```
public AB espejo ()
{
        AB arbol = new AB();
        arbol.raiz = espejo(this.raiz);
        return arbol;
}

private Nodo espejo (Nodo nodo) { }
```



2) **(10 puntos)** El método **esArbolDegenerado** que determina si un árbol binario es degenerado. Un árbol degenerado es aquel que cada nodo contiene solo un hijo.

```
public boolean esArbolDegenerado()

private boolean esArbolDegenerado(Nodo nodo)
```



Tema 6 (25 puntos)

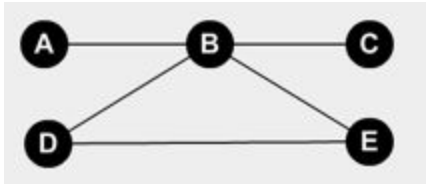
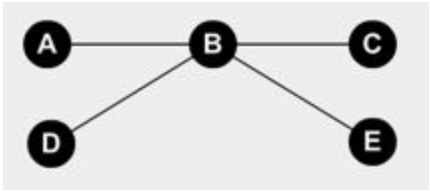
Para la resolución de los siguientes ejercicios de grafos consulte las referencias del TDA Grafo.

a) **(10 puntos)** Implemente el método **existeRecorridoEuleriano**, que recibe un grafo y retorna **true** si es posible realizar un recorrido euleriano en el grafo. El recorrido euleriano es aquel que visita todos los arcos solo una vez. Un recorrido euleriano es posible, si y sólo si:

i) El grafo es conexo

ii) Si el grafo tiene a lo mucho hasta 2 vértices de grado impar. El grado es el número de arcos que tiene un vértice.

```
public boolean existeRecorridoEuleriano(Grafo<T> g){ … }
```

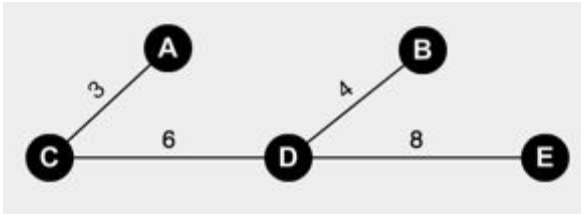
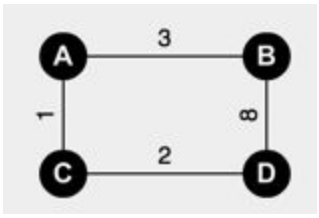
	
Si existe recorrido euleriano	No existe recorrido euleriano

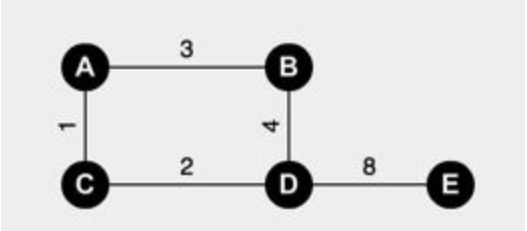
b) **(15 puntos)** Implemente el método **unirGrafos**, que recibe dos grafos no dirigidos, y retorna un nuevo grafo con la unión de los mismos de acuerdo a las siguientes reglas:

i) El grafo resultante deberá contener tanto los vértices (contenidos) del grafo G1 y G2.

ii) El grafo deberá **<u>unir</u>** los mismos vértices que G1 y G2, en caso de que existan dos arcos que unan un mismo par de vértices, se tomará el arco con menor tamaño.

```
public Grafo<T> unirGrafos(Grafo<T> g1, Grafo<T> g2) { … }
```

	
Grafo G1	Grafo G2


Grafo resultante

Referencias

public class HashMap<K,V>	
HashMap()	Constructor, inicializa el hashmap.
boolean containsKey(Object key)	Retorna true si el map contiene la clave
V get(Object key)	Obtiene el valor asociado con la clave.
V put(K key, V value)	Agrega una entrada al hashmap con la clave y valor pasados como parámetros.
V remove(Object key)	Remueve la clave y el valor del hashmap/
Set<K> keySet()	Retorna un Set con todas las claves.
Set<Map.Entry<K,V>> entrySet()	Retorna un Set con un par clave-valor
Collection<V> values()	Retorna una colección con todos los valores del hashmap.
public class Grafo<T>	
Grafo(boolean dirigido)	Constructor, inicializa el grafo. Si dirigido es true creará un Grafo Dirigido caso contrario será un Grafo No Dirigido.
void agregarVertice(T contenido)	Agrega un nuevo vértice al grafo
void agregarArco(T origen, T destino) void agregarArco(T origen, T destino, int peso)	Agrega un arco entre dos vértices. Si se le especifica el peso, entonces ese será el peso del arco. Caso contrario, el peso será de 1.
List<Vertice<T>> getVertices()	Retorna una lista con todos los vértices del grafo.
Vertice<T> buscarVertice(T contenido)	Retorna el vértice con el contenido especificado, si no existe retorna null.
Arco<T> buscarArco(T origen, T destino)	Retorna true si existe un arco entre origen y destino.
void removerVertice(T contenido)	Remueve el vértice con el contenido especificado, asimismo que todos los arcos que tiene.
void removerArco(T origen, T destino)	Remueve el arco con el origen y destino especificados.
List<Vertice> dfs(T inicio)	Hace un recorrido en profundidad a partir del vértice con el contenido inicio.

List<Vertice> bfs(T inicio)	Hace un recorrido en anchura partir del vértice con el contenido inicio.
int componentesConexas()	Retorna el número de componentes conexas del grafo.
void resetearVisitados()	Establece todos los vértices del grafo como no visitados.
public class Vertice<T>	
Vertice(T contenido)	Constructor
T getContenido()	Obtiene el contenido del vértice
void setContenido(T contenido)	Establece el contenido del vértice
boolean estaVisitado()	Obtiene si el vértice ha sido visitado o no
void setVisitado(boolean visited)	Establece un vértice como visitado
List<Arco<T>> getArcos()	Retorna la lista de arcos del vértice.
public class Arco<T>	
Arco(Vertice origen, Vertice destino, int peso)	Constructor
Vertice<T> getOrigen()	Obtiene el vértice de origen
void setOrigen(Vertice<T> origen)	Establece el vértice de origen
Vertice<T> getDestino()	Obtiene el vértice de destino
void setDestino(Vertice<T> destino)	Establece el vértice de destino
int getPeso()	Obtiene el peso del arco
void setPeso(int peso)	Establece el peso del arco