## Recuperación Segundo parcial de EDA del 11 de Junio de 2013 (Duración: 2h)

| APELLIDOS, NOMBRE |
| --- |
| |

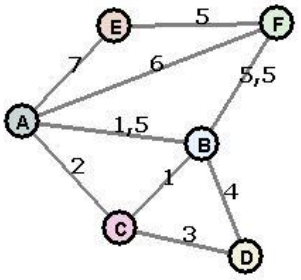
**1.-** Completa la clase *MonticuloBinario*, que representa un *heap* minimal, con un método que elimine la primera hoja del montículo. **(2.5 puntos)**

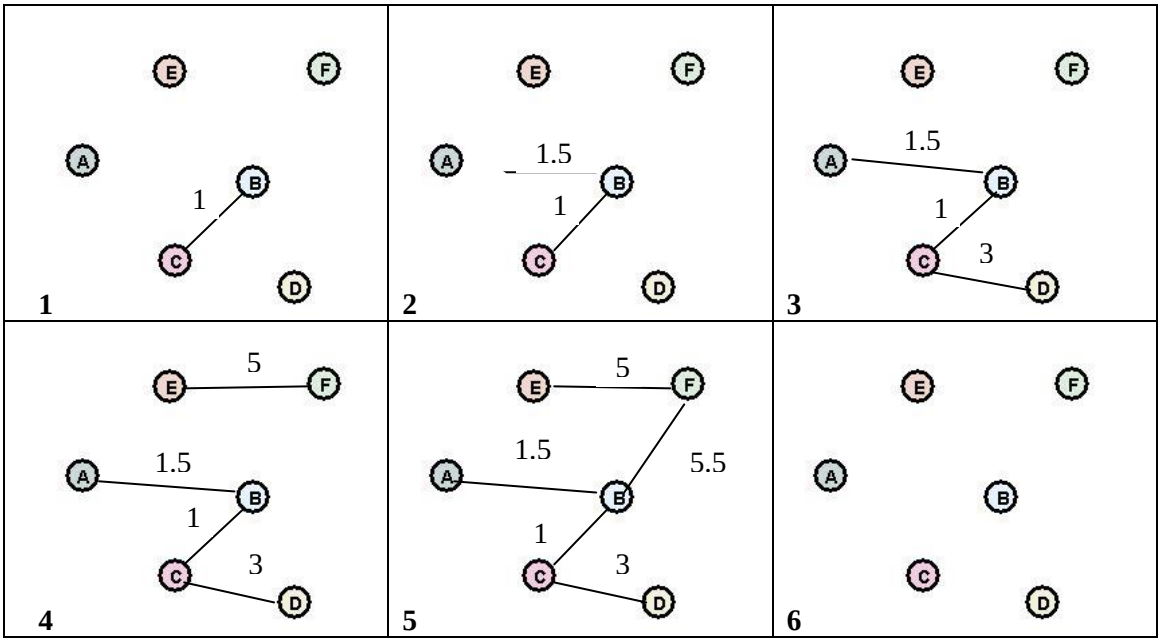
```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
    protected E elArray[];
    protected static final int CAPACIDAD_POR_DEFECTO = 11;
    protected int talla;

    public void eliminarprimeraHoja() {
       int primeraHoja = (talla/2)+1;
       elArray[primeraHoja] = elArray[talla--];
       int i = primeraHoja; E eEnI = elArray[primeraHoja];
       while ( i>1 && eEnI.compareTo(elArray[i/2])<0 ){
          elArray[i] = elArray[i/2]; i /= 2;
       }
       elArray[i] = eEnI;
    }


}
```

**2.-** Se dice que un vértice de un grafo Dirigido es un Sumidero si su grado de entrada es mayor que cero y su grado de salida es cero. Se pide diseñar en la clase `GrafoDirigido` un método que, en tiempo constante en el Mejor de los Casos y en tiempo lineal con el nº de aristas en el Peor, compruebe si el vértice `v` es Sumidero de todos los de un Grafo, es decir que sobre él inciden todos los demás y su grado de salida es 0. **(3 puntos)**

```
public class GrafoDirigido extends Grafo {
    protected int numV, numA;
    protected ListaConPI<Adyacente> elArray[];

    public GrafoDirigido(int numVertices){…}

public boolean esSumidero(int v) {
   if (elArray[v].talla() !=0 ) return false;
   for( int i=0; i<numV; i++ ) {
       if (i!=v){
           ListaConPI<Adyacente> l = elArray[i];
           boolean enc = false;
           for (l.inicio(); !l.esFin() && !enc; l.siguiente())
               if (l.recuperar().destino==v) enc=true;
           if (!enc) return false;
       }
   return true;
}









}
```

**3.-** Completar las figuras indicando las aristas que va incorporando el algoritmo de Kruskal hasta obtener el árbol de expansión de coste mínimo (Minimun Spanning Tree) para el siguiente Grafo: **(2 puntos)**




**4.** Se tiene una *ListaConPI* con información sobre tramos de carretera y su fecha de finalización, ordenados de forma creciente según esta fecha. Se pide diseñar el método *fechaParaLlegar* que, dada esta lista y dos pueblos x e y devuelva la fecha a partir de la cual se puede viajar de uno a otro. Si no es posible llegar de x a y devolverá null. Se supone que puede haber 200 pueblos distintos con códigos entre 0 y 199. **(2.5 puntos)**

```
public static String fechaParaLlegar(int x, int y, ListaConPI<Tramo> l) {
  MFSet mfs = new ForestMFSet(200);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
     Tramo t = l.recuperar();
     String fecha = t.getFecha();
     mfs.merge(t.getPueblo1(), t.getPueblo2());
     if (mfs.find(x)==mfs.find(y)) return fecha;
  }
  return null;
}
```

Supóngase definida la clase Tramo como sigue:

```
public class Tramo{
   private int pueblo1, pueblo2;
   private String fecha;
   public Tramo(int p1, int p2, String f){...}
   public int getPueblo1(){...}
   public int getPueblo2(){...}
   public String getFecha(){...}
}
```