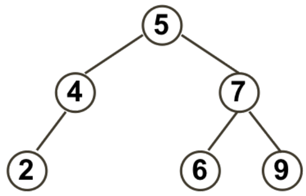
Resolución del Segundo Parcial de EDA (8 de Junio de 2018)

Problema 1 (ABB). 2.5 puntos



a) (2.0 puntos) Sea la clase **`ABBInteger`** una derivada de **`ABB`**. Diseña en ella un método **`caminoQueSuma`** que, con el menor coste posible y recursivamente, compruebe si existe una secuencia de nodos (camino) desde su raíz a uno de sus descendientes cuyos datos sumen **`s`**, con **`s`**`>0`. Así, por ejemplo, dado el ABB de la derecha, el método devuelve `true` si el valor de `s` es `9`, `11` o `12`; sin embargo, devuelve `false` si el valor de `s` es `10` o `19`.

Para diseñar este método solo puedes acceder a los atributos de la clase `ABB`, que `ABBInteger` hereda. Además, para simplificar, supón como **precondición** que el árbol sobre el que se aplica el método está Equilibrado, no es vacío y que el dato que ocupa su raíz es menor estricto que `s`.

```java
public class ABBDeInteger extends ABB<Integer> {
    public ABBDeInteger() { super(); }

    public boolean caminoQueSuma(int s) {
        return caminoQueSuma(s, this.raiz);
    }
    protected boolean caminoQueSuma(int s, NodoABB<Integer> actual) {
        if (actual == null) { return false; }
        s -= actual.dato;
        if (s == 0) { return true; }  // Encontrado
        if (s < 0) { return false; }  // La suma del camino supera la buscada
        if (s <= actual.dato) { return caminoQueSuma(s, actual.izq); }
        return caminoQueSuma(s, actual.izq) || caminoQueSuma(s, actual.der);
    }
}
```

b) (0.5 puntos) Indica para el método recursivo diseñado: la talla del problema **x** que resuelve, en función de sus parámetros; la relación o relaciones de recurrencia que definen su coste temporal; utilizando la notación asintótica (O y Ω o bien Θ), su coste temporal.

Talla del problema, en función de los atributos de la clase: **x** = `talla(actual)`, el tamaño del nodo actual

Relaciones de Recurrencia: en el caso general, cuando **x > 0**,

- En el Mejor de los Casos, cuando por ejemplo el camino que suma `s` tiene longitud 1,

  $T^{M}_{\texttt{caminoQueSuma}}(x) = k.$

- En el Peor Caso, cuando por ejemplo s es mayor que la suma de todos los datos del ABB,

  $T^{P}_{\texttt{caminoQueSuma}}(x) = 2 * T^{P}_{\texttt{caminoQueSuma}}(x / 2) + k'.$

Utilizando la notación asintótica, el coste temporal del método recursivo diseñado es:

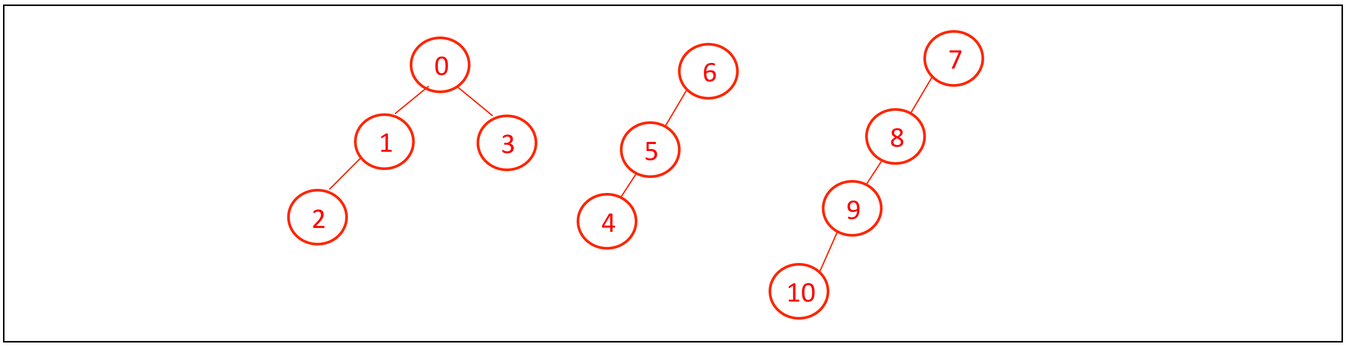
$T_{\texttt{caminoQueSuma}}(x) \in \Omega(1)$ y $T_{\texttt{caminoQueSuma}}(x) \in O(x).$

Problema 2 (MF-Sets). 1.5 puntos

Sea la siguiente representación de un **MF-Set**:

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
-3	0	1	0	5	6	-3	-4	7	8	9

a) (0.5 puntos) Dibuja el bosque de árboles que contiene.

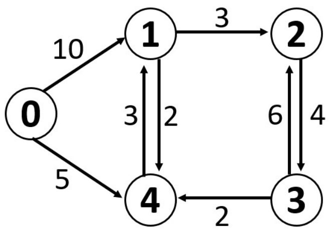


b) (1.0 punto) Teniendo en cuenta que se implementa la fusión por rango y la compresión de caminos, indica cómo irá evolucionando dicha representación tras la ejecución de las instrucciones de la siguiente tabla. **Nota:** Al unir dos árboles con el mismo rango, el primero ha de colgar del segundo.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
`find(10)`	-3	0	1	0	5	6	-3	-4	7	7	7
`merge(6,7)`	-3	0	1	0	5	6	7	-4	7	7	7
`merge(0,6)`	7	0	1	0	5	6	7	-4	7	7	7

Problema 3 (Grafos). 2 puntos

Haz una traza del algoritmo de **Dijkstra** para el siguiente grafo, tomando como vértice origen el 0. Rellena tantas filas como sean necesarias.



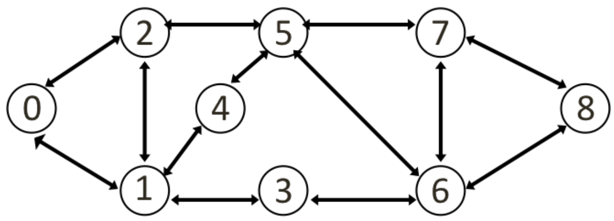
		visitados					*distanciaMin*					*caminoMin*				
v	*qPrior*	*0*	*1*	*2*	*3*	*4*	*0*	*1*	*2*	*3*	*4*	*0*	*1*	*2*	*3*	*4*
	(0, 0)	0	0	0	0	0	0	∞	∞	∞	∞	-1	-1	-1	-1	-1
0	(1,10) (4,5)	1	0	0	0	0	0	10	∞	∞	5	-1	0	-1	-1	0
4	(1,10) (1,8)	1	0	0	0	1	0	8	∞	∞	5	-1	4	-1	-1	0
1	(1,10) (2,11)	1	1	0	0	1	0	8	11	∞	5	-1	4	1	-1	0
1	(2,11)	1	1	0	0	1	0	8	11	∞	5	-1	4	1	-1	0
2	(3,15)	1	1	1	0	1	0	8	11	15	5	-1	4	1	2	0
3		1	1	1	1	1	0	8	11	15	5	-1	4	1	2	0

Problema 4 (Grafos). 4 puntos

En la clase **Grafo** se quiere implementar, para un grafo no ponderado, un método de instancia **verticesCercanos** que, dado un vértice **v** y una distancia **n**, devuelva un `String` que contenga los vértices alcanzables desde **v** con una distancia máxima **n**, indicando también para cada vértice alcanzable su distancia (al ser el grafo no ponderado, la distancia entre dos vértices es igual al número de aristas que los conectan, dado que todas tienen peso 1.0).

Si **g** es el grafo cuya imagen se muestra a la derecha, considerar el siguiente bloque de código:

```
System.out.println(g.verticesCercanos(0,1));
System.out.println(g.verticesCercanos(0,2));
System.out.println(g.verticesCercanos(0,3));
```



Cuya salida debería ser la siguiente:

```
[(1, 1) (2, 1) ]
[(1, 1) (2, 1) (3, 2) (4, 2) (5, 2) ]
[(1, 1) (2, 1) (3, 2) (4, 2) (5, 2) (6, 3) (7, 3) ]
```

La implementación de **verticesCercanos** debe ser eficiente, garantizando que no se visitarán los vértices que se encuentren a una distancia mayor que **n** (a más de **n** aristas de **v**).

```
public String verticesCercanos(int v, int n) {

    distanciaMin = new double[numVertices()];      // Se Inicializa la distancia
    for (int i = 1; i < numVertices(); i++) {       // mínima desde v a todos los
        distanciaMin[i] = INFINITO;                  // vértices del grafo, ...
    }
    distanciaMin[v] = 0;                             // v incluido

    q = new ArrayCola<Integer>();          // Se crea la Cola q vacía,
    q.encolar(v);                          // se encola el vértice v
    String res = "[";                      // y se inicializa el String resultado

    while (!q.esVacia()) {                 // Se realiza el Recorrido BFS de this Grafo
        int u = q.desencolar();            // Se desencola el vértice u a recorrer en la iteración
        if (distanciaMin[u] < n) {         // Si la distancia desde v hasta u aún NO excede n, la máxima:
            ListaConPI<Adyacente> l = adyacentesDe(u);
            for (l.inicio(); !l.esFin(); l.siguiente()) {   // Se recorre cada vértice
                int w = l.recuperar().getDestino();          // w adyacente a u que
                if (distanciaMin[w] == INFINITO) {           // aún NO se haya visitado,
                                                             // actualizando:
                    distanciaMin[w] = distanciaMin[u] + 1;   // su distancia a v,
                    res += "(" + w + ", "                    // el String resultado
                           + (int)distanciaMin[w] + ") ";
                    q.encolar(w);                            // y la Cola q
                }
            }
        }
    }
    return res.trim() + "]";               // Se devuelve el String resultado
}
```

ANEXO

Las clases `ABB` y `NodoABB` del paquete `jerarquicos`.

```
public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;
    public ABB() { this.raiz = null; }
    ...
}
```

```
class NodoABB<E> {
    protected E dato;
    protected NodoABB<E> izq, der;
    protected int talla;
    NodoABB(E e) {
        this.dato = e;
        this.izq = null; this.der = null;
        this.talla = 1;
    }
}
```

La clase `Grafo` del paquete `grafos`.

```
public abstract class Grafo {
     protected static final double INFINITO = Double.POSITIVE_INFINITY;
     protected boolean esDirigido;
     // atributos "auxiliares"
     protected int[] visitados;
     protected int ordenVisita;
     protected Cola<Integer> q;
     protected double[] distanciaMin;
     protected int[] caminoMin;
     ...
     public abstract int numVertices();
     public abstract int numAristas();
     public abstract ListaConPI<Adyacente> adyacentesDe(int i);
     ...
}
```

Teoremas de coste

Teorema 1: $f(x) = a \cdot f(x - c) + b$, con $b \geq 1$

- si a=1, $f(x) \in \Theta(x)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$;

Teorema 2: $f(x) = a \cdot f(x - c) + b \cdot x + d$, con b y d≥1

- si a=1, $f(x) \in \Theta(x^2)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$;

Teorema 3: $f(x) = a \cdot f(x/c) + b$, con $b \geq 1$

- si a=1, $f(x) \in \Theta(\log_c x)$;
- si a>1, $f(x) \in \Theta(x^{\log_c a})$;

Teorema 4: $f(x) = a \cdot f(x/c) + b \cdot x + d$, con b y d≥1

- si a<c, $f(x) \in \Theta(x)$;
- si a=c, $f(x) \in \Theta(x \cdot \log_c x)$;
- si a>c, $f(x) \in \Theta(x^{\log_c a})$;

Teoremas maestros

Teorema para recurrencia divisora: la solución a la ecuación $T(n) = a \cdot T(n/b) + \Theta(n^k)$, con a≥1 y b>1 es:

- $T(n) = O(n^{\log_b a})$ si $a>b^k$;
- $T(n) = O(n^k \cdot \log n)$ si $a=b^k$;
- $T(n) = O(n^k)$ si $a<b^k$;

Teorema para recurrencia sustractora: la solución a la ecuación $T(n) = a \cdot T(n-c) + \Theta(n^k)$ es:

- $T(n) = \Theta(n^k)$ si a<1;
- $T(n) = \Theta(n^{k+1})$ si a=1;
- $T(n) = \Theta(a^{n/c})$ si a>1;