Primer Parcial de Estructuras de Datos y Algoritmos (EDA) – 28 de Marzo de 2022 - Duración: 1h. 30m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(2,25 puntos)*** Implementa un método estático tal que:

- Reciba una **`ListaConPI`** genérica, cuyos elementos están ordenados ascendentemente, y que puede contener valores repetidos.
- Borre de la lista recibida todos los valores repetidos.
- Devuelva el número de elementos borrados.

La solución debe ser eficiente (coste lineal con la talla de la lista), y no se debe usar ninguna EDA auxiliar.

Ejemplos.

- Si se recibe la lista [1,1,2,2,2,3,3,4,4,5,7,7,7,8], el método devolverá 7 y el contenido de la lista cambiará a [1,2,3,4,5,7,8].
- Si se recibe la lista [“aa”,”aa”,”ab”,”ba”,”bb“,“bb”,”bc”,”cc”,”cc”,”cc”], el método devolverá 4 y el contenido de la lista cambiará a [“aa”,”ab”,”ba”,”bb“,”bc”,”cc”].

```
public static <E> int metodo1(ListaConPI<E> l) {
    int i = 0;
    l.inicio();
    // bucle de recorrido de la lista
    while (!l.esFin()) {
        E e = l.recuperar();
        l.siguiente();
        // bucle de búsqueda, y borrado, de valores repetidos
        // que solo pueden estar en posiciones consecutivas
        while (!l.esFin()) {
            E f = l.recuperar();
            if (f.equals(e)) { l.eliminar(); i++; }
            else break;
        }
    }
    return i;
}
```

2.- ***(2,50 puntos)*****.** Aplicación de la estrategia **Divide y Vencerás (DyV)**.

Sea **v**, un array genérico de elementos **`Comparable`**, ordenado ascendentemente y que puede contener valores repetidos, y sea **e**, un elemento de la misma clase que los elementos de **v**. Se quiere obtener el número de apariciones de **e** en **v**.

Aplicando DyV, se pide implementar dos métodos (uno lanzadera y otro recursivo) que devuelvan el número de apariciones de **e** en **v**.

El problema tiene instancias significativas. Con una correcta aplicación de DyV, el caso mejor, y el caso promedio, tendrán coste logarítmico. Aunque existe un caso peor de coste lineal (cuando el elemento buscado **e** está repetido en todas las posiciones del array **v**).

Algunos ejemplos en la siguiente tabla:

v	**e**	*resultado*
[1,1,2,2,2,3,3,4,4,5,7,7,7,8,9,9,9]	3	2
[1,1,2,2,2,3,3,4,4,5,7,7,7,8,9,9,9]	6	0
[1,1,2,2,2,3,3,4,4,5,7,7,7,8,9,9,9]	7	3
[1,1,2,2,2,3,3,4,4,5,7,7,7,8,9,9,9]	8	1
[“aa”,”aa”,”ab”,”ba”,”bb“,“bb”,”bc”,”cc”,”cc”,”cc”,”da”,”da”]	“aa”	2
[“aa”,”aa”,”ab”,”ba”,”bb“,“bb”,”bc”,”cc”,”cc”,”cc”,”da”,”da”]	“cc”	3

```
public static <E extends Comparable <E>> int metodo2(E[] v, E e) {
    return metodo2(v, e, 0, v.length - 1);
}

private static <E extends Comparable <E>> int metodo2(E[] v, E e, int i, int f) {
    // caso base, subarray vacío
    if (i > f) return 0;
    int m = (i + f) / 2;
    int comp = v[m].compareTo(e);
    // si v[m] es mayor, buscar en mitad inferior
    if (comp > 0) return metodo2(v, e, i, m - 1);
    // si v[m] es menor, buscar en mitad superior
    else if (comp < 0) return metodo2(v, e, m + 1, f);
    // si v[m] es igual a e, contarlo, y buscar en ambas mitades
    else return 1 + metodo2(v, e, i, m - 1) + metodo2(v, e, m + 1, f);
}
```

3.- ***(2,75 puntos)*** Durante unos disturbios, la policía ha monitorizado todas las comunicaciones en la zona y dispone de una **`ListaConPI`** de **`String`** (conexiones, identificadas mediante números de móvil), con todas las conexiones realizadas. La policía estima que aquellos números de móvil que hayan hecho más de **`umbral`** conexiones podrían corresponder a los cabecillas de los disturbios. La policía también estima que el número de cabecillas es, como máximo, un 10% del número total de conexiones.

Para identificar a los cabecillas, se quiere implementar un método estático que reciba la **`ListaConPI`** de **`String`**, con todas las conexiones, y el **`umbral`** (un entero), y que, eficientemente (en el menor tiempo posible), devuelva otra **`ListaConPI`** de **`String`**, con todos los números de móvil que hayan realizado al menos el número de conexiones establecidas por dicho **`umbral`**. Los titulares de esos móviles serán, probablemente, los cabecillas de los disturbios.

```
public static ListaConPI<String> metodo3(ListaConPI<String> l, int umbral) {
    ListaConPI<String> res = new LEGListaConPI<String>();
    // map auxiliar, claves: teléfonos o conexiones,
    // valores: número de llamadas o conexiones de cada teléfono
    Map<String,Integer> m = new TablaHash<String,Integer>( l.talla()/10 );
    // bucle de recorrido de la lista
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        // recuperar teléfono, que será clave del map
        String tel = l.recuperar();
        // dada la clave, recuperar su valor asociado (número de conexiones)
        Integer num = m.recuperar(tel);
        // inicializar o incrementar el valor
        if (num == null) num = 1;
        else num++;
        // insertar, en map, la clave con su valor
        m.insertar(tel, num);
        // insertar, en lista resultado, la clave si el valor es igual a umbral
        if (num == umbral) { res.insertar(tel); }
    }
    return res;
}
```

Comentarios adicionales:

- 1) La inserción en lista resultado se podría hacer con otro bucle, posterior, donde se recorriera todo el map (sus claves) y se encontraran las claves con valor asociado mayor o igual a umbral. Esta solución alternativa, aunque algo menos eficiente, sigue teniendo coste lineal con la talla de la lista recibida, y se considera válida para este ejercicio.

- 2) En el enunciado, donde se dice: "el número de cabecillas es, como máximo, un 10% del número total de conexiones", hubiera sido más adecuado decir "el número de participantes en los disturbios", o "el número de teléfonos distintos". Para así dimensionar el map auxiliar. Por ello, se considerará válido en este ejercicio dimensionar el map en cualquier valor entre l.talla()/10 y l.talla().

- 3) En el enunciado, se dice: "hayan hecho más de umbral conexiones", y también se dice: "hayan realizado al menos el número de conexiones establecidas por dicho umbral". Hay una contradicción. Por ello, se considerará válida también la solución donde, para insertar en lista resultado, se compare con umbral+1 en lugar de con umbral.

4.- ***(2,50 puntos)*** Implementa en la clase **MonticuloBinario**, un método que elimine del montículo aquellos elementos para los que un método auxiliar **valido** devuelve **false**. Dicho método, que se supone implementado, tiene la siguiente cabecera:

```
private boolean valido(E e);
```

La clase **MonticuloBinario** representa un montículo minimal. Por tanto, el objeto modificado al ejecutar el método pedido deberá seguir siendo un montículo minimal (es decir, se deben preservar sus propiedades estructural y de orden).

Se debe implementar con el menor coste y sin usar ninguna EDA auxiliar.

```
public void metodo4() {
    int i = 1;
    // bucle de recorrido de todo el montículo
    while (i <= talla) {
        // los elementos "no válidos" se borran preservando propiedad estructural
        if (!valido(elArray[i])) { elArray[i] = elArray[talla--]; }
        else { i++; }
    }
    // bucle de "hundir" nodos internos, no-hojas, para restablecer propiedad de orden
    for (i = talla / 2; i > 0; i--) { hundir(i); }
}
```

ANEXO.

Interfaces **ListaConPI, ColaPrioridad** y **Map** del paquete **modelos**. Clase **MonticuloBinario** del paquete **jerarquicos**.

```
public interface ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  void inicio();
  void siguiente();
  void fin();
  E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}

public interface ColaPrioridad<E extends Comparable<E>> {
  void insertar(E e);
  E eliminarMin();
  E recuperarMin();
  boolean esVacia();
}

public interface Map<C, V> {
  V insertar(C c, V v);
  V eliminar(C c);
  V recuperar(C c);
  boolean esVacio();
  int talla();
  ListaConPI<C> claves();
}

public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E>
{
  protected E[] elArray;
  protected int talla;
  ...
  protected void hundir(int pos) { ... }
  ...
}
```