Examen 20 Enero 2016, preguntas y respuestas

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

Resolución de la Recuperación del Primer Parcial de EDA (20 de Junio de 2016) - Puntuación: 3

1.- Una fábrica dispone de una máquina que prepara bombones y los almacena en cajas con un número par de unidades. Aunque la máquina fabrica a veces un bombón defectuoso, que tiene un peso mayor que el resto, se puede asegurar que no puede haber más de uno de estos bombones en cada caja. A nivel de programación, una caja `c` se puede representar como un array de objetos de tipo `Bombon` (`Bombon[] c`); además, es posible conocer el peso de un subconjunto de bombones de una caja (subarray `c[i, j]`) en tiempo constante mediante el siguiente método:

```
// SII i>= 0 && i<=j && j<caja.length: devuelve el peso de los bombones situados entre
// las posiciones i y j, ambas inclusive, de una caja c
public static float peso(Bombon[] c, int i, int j) { … }
```

Diseña un método estático Divide y Vencerás que devuelva la posición que ocupa el bombón defectuoso en una caja `c`, o -1 si la caja no contiene tal bombón. El método diseñado debe tener un coste temporal O(log `c.length`). **(1 punto)**

```
public static int bombonDefectuoso(Bombon[] c) {
    // Al ser par el nº de bombones por caja...
    // si las 2 mitades de la caja c pesan lo mismo, no hay bombón defectuoso;
    // sino, buscar el bombón defectuoso **con garantía de éxito**
    int m = c.length / 2 – 1;
    if (peso(c, 0, m) == peso(c, m+1, c.length - 1)) { return -1; }
    else { return bombonDefectuoso(c, 0, c.length – 1); }
}

// Búsqueda con garantía de éxito: seguro que hay un bombón defectuoso en c[i, j]
private static int bombonDefectuoso(Bombon[] c, int i, int j) {
    // Caso base: si c tiene un único bombón, seguro que es defectuoso
    if (i == j) { return i; }

    // Caso general: si c tiene 2 o más bombones, usar DyV para buscar el defectuoso

    // (a) DIVIDIR la caja c[i, j] en 2 mitades
    int m = (i + j) / 2, talla = j – i + 1;
    float pesoIzq, pesoDer;

    // (b) VENCER, en base al peso que tenga cada mitad de c[i, j]
    //     OJO: según la paridad de la talla de c[i, j], c[m] está
    //          en una de las mitades a pesar o las separa
    if (talla % 2 == 0) {
    // talla de c[i, j] es par → c[m] es el último bombón de la 1era mitad a pesar
        pesoIzq = peso(c, i, m);
        pesoDer = peso(c, m + 1, j);
        if (pesoIzq > pesoDer) { return bombonDefectuoso(c, i, m); }
        else { return bombonDefectuoso(c, m + 1, j); }
    }
    else {
    // talla de c[i, j] es impar → c[m] es el bombón que separa las mitades a pesar
        pesoIzq = peso(c, i, m - 1);
        pesoDer = peso(c, m + 1, j);
        if (pesoIzq == pesoDer) { return m; }
        else if (pesoIzq > pesoDer) { return bombonDefectuoso(c, i, m - 1); }
        else { return bombonDefectuoso(c, m + 1, j); }
    }
}
```

2.- Diseña un método estático que, dado un array genérico `E[] v`, devuelva una *Lista con Punto de Interés* con aquellos elementos de `v` cuya frecuencia de aparición sea menor que un umbral dado `u` (`int`). Para lograr un coste temporal O(`v.length`) tu método debe usar un *Map* implementado mediante una *Tabla Hash*. **(0.75 puntos)**

```
public static ListaConPI<E> menorQue(E[] v, int u) {
    Map<E, Integer> map = new TablaHash<E, Integer>(v.length);
    ListaConPI<E> res = new LEGListaConPI<E>();
    for (int i = 0; i < v.length; i++) {
        Integer frec = map.recuperar(v[i]);
        if (frec == null) frec = 0;
        map.insertar(v[i], frec + 1);
    }
    ListaConPI<E> l = map.claves();
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Integer frec = map.recuperar(l.recuperar());
        if (frec < u) res.insertar(l.recuperar());
    }
    return res;
}
```

3.- Sea un *ABB* sin elementos repetidos.

a) Implementa en la clase `ABB` un método recursivo que, con el mejor coste Temporal posible, devuelva una *Lista con Punto de Interés* con los elementos del *ABB* que hay en el camino desde su raíz hasta un cierto elemento `x`, o `null` si dicho elemento no está en el *ABB*. **(0.75 puntos)**

```
public ListaConPI<E> camino(E x) {
    ListaConPI<E> res = new LEGListaConPI<E>();
    return camino(raiz, x, res);
}

protected ListaConPI<E> camino(NodoABB<E> actual, E x, ListaConPI<E> res) {
    if (actual == null) { return null; }
    res.insertar(actual.dato);
    int cmp = actual.dato.compareTo(x);
    if (cmp > 0) { return camino(actual.izq, x, res); }
    else if (cmp < 0) { return camino(actual.der, x, res); }
    return res;
}
```

(b) Estudia el coste del método diseñado para un *ABB* Equilibrado. **(0.5 puntos)**

Talla: **n =** tamaño del nodo `actual`, el del ABB en la llamada más alta.

Instancias significativas:

Mejor caso: en la llamada más alta, el dato del nodo `actual` es igual a `x`.

Peor caso: `x` no está en el *ABB*.

Ecuaciones de Recurrencia:

$T_{camino}^{P}(n = 0) = k2$; $T_{camino}^{P}(n > 0) = 1 * T_{camino}(n/2) + k3$

Coste Temporal Asintótico:

$T_{camino}(n) \in \Omega(1)$ y $T_{amino}(n) \in O(\log_2 n)$

ANEXO

```
public interface ListaConPI<E> {
    void insertar(E e);
    /** SII !esFin() */ void eliminar();
    void inicio();
    /** SII !esFin() */ void siguiente();
    void fin();
    /** SII !esFin() */ E recuperar();
    boolean esFin();
    boolean esVacia();
    int talla();
}
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 TablaHash<C,V> implements Map<C,V> {
    …
    public TablaHash(int inicial) {…}
    …
}
```

```
public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;
    public ABB() {…}
    public boolean esVacio() {…}
    public int talla() {…}
    public E recuperar(E e) {…}
    public void insertar(E e) {…}
    public void eliminar(E e) {…}
    …
}
class NodoABB<E> {
    E dato;
    int talla;
    NodoABB<E> izq, der;
    NodoABB(E e) {…}
    NodoABB(E e, NodoABB<E> i, NodoABB d) {…}
}
```

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 $d\geq 1$
- 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 $d\geq 1$
- 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 $a\geq 1$ 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 $a>1$;