Ejercicio de seminario - tercer parcial

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

Tercer parcial

Heaps, Tablas Hash y Grafos

EJERCICIOS DE REPASO

Ejercicio 1.- Escribe un método en la clase *MonticuloBinario* que permita sustituir el elemento que está en una posición dada por otro:

```
public void sustituir(int pos, E x);
```

¿Qué coste tiene este método?

Solución:

```
public void sustituir(int pos, E x) {
  if (x.compareTo(elArray[pos]) > 0) {
    elArray[pos] = x;
    hundir(pos);
  } else {
    while (pos > 1 && x.compareTo(elArray[pos/2]) < 0) {
      elArray[pos] = elArray[pos/2];
      pos = pos/2;
    }
    elArray[pos] = x;
  }
}
```

El coste de este método es:

$T_{sustituir}^{M}(talla) \in \Omega(1)$
$T_{sustituir}^{P}(talla) \in O(\log_2 talla)$
, donde talla = tamaño del montículo.

Ejercicio 2.- Diseña un método recursivo dentro de la clase *MonticuloBinario* que permita contar el número de apariciones de un elemento dado dentro del montículo:

```
public int numApariciones(E x);
```

Analiza el coste de este método.

Solución:

```
public int numApariciones(E x) {
  return numApariciones(x, 1);
}
```

```
private int numApariciones(E x, int pos) {
  if (pos > talla) return 0;
  int resC = elArray[pos].compareTo(x);
  if (resC > 0) return 0;
  int n = numApariciones(x, 2*pos) + numApariciones(x, 2*pos + 1);
  if (resC == 0) n++;
  return n;
}
```

Análisis del coste:

Talla: n = tamaño del nodo que ocupa la posición *pos* (tamaño del montículo en la primera llamada)

Instancias significativas:

- Mejor caso: elArray[pos] es mayor que x
- Peor caso: x es mayor o igual que todos los elementos del montículo

Relaciones de recurrencia:
$T_{numApariciones}^{M}(n) = k_1$
$T_{numApariciones}^{P}(n = 0) = k_2$
$T_{numApariciones}^{P}(n > 0) = 2 * T_{numApariciones}^{P}(n/2) + k_3$

Coste temporal asintótico:
$T_{numApariciones}(n) \in \Omega(1)$
$T_{numApariciones}(n) \in O(n)$, por el teorema 3 con a=c=2

Ejercicio 3.- Dado un vector genérico de objetos comparables (cuyos datos comienzan en la posición 1), diseña un método recursivo que compruebe de la forma más eficiente posible si sus datos cumplen la propiedad de orden de un Montículo Minimal. Estudia el coste del método diseñado.

Solución:

```
public <E extends Comparable<E>> boolean esHeap(E v[]) {
  return esHeap(v, 1);
}

public <E extends Comparable<E>> boolean esHeap(E v[], int pos) {
  if (pos > talla) return true;
  int izq = pos * 2;
  if (izq <= talla && elArray[i].compareTo(elArray[izq]) > 0) return false;
  if (izq < talla && elArray[i].compareTo(elArray[izq+1]) > 0) return false;
  return esHeap(v, izq) && esHeap(v, izq+1);
}
```

Análisis del coste:

Talla: n = tamaño del nodo que ocupa la posición *pos* (tamaño del montículo en la primera llamada)

Instancias significativas:

- Mejor caso: elArray[pos] es mayor que elArray[pos*2]
- Peor caso: el array *v* cumple la propiedad de orden de un montículo (minimal), es decir, que para cualquier nodo, su valor es menor o igual que el de sus hijos.

Relaciones de recurrencia:
$T_{esHeap}^{M}(n) = k_1$
$T_{esHeap}^{P}(n = 0) = k_2$
$T_{esHeap}^{P}(n > 0) = 2 * T_{esHeap}^{P}(n/2) + k_3$

Coste temporal asintótico:
$T_{esHeap}(n) \in \Omega(1)$
$T_{esHeap}(n) \in O(n)$, por el teorema 3 con a=c=2

Ejercicio 4.- Supóngase que se ha modificado la clase *TablaHash<E>* para permitir la inserción de elementos duplicados. Se pide:

1. Añadir el método *recuperarIguales* a la clase *TablaHash<E>* que devuelva una Lista con Punto de Interés con todos los elementos que sean iguales a uno dado.
2. Añadir el método *recuperarValores* a la clase *TablaHashDiccionario<C, V>* que, haciendo uso del método anterior, devuelva una Lista con Punto de Interés que contenga todos valores asociados a una clave dada.

Solución:

```
public ListaConPI<E> recuperarIguales(E x) {
  ListaConPI<E> res = new LEGListaConPI<E>();
  // todos los elementos de la Tabla iguales a x sólo pueden estar en una
  //  cubeta: elArray[posTabla(x)]
  ListaConPI<E> lpi = elArray[posTabla(x)];
  for ( lpi.inicio(); !lpi.esFin(); lpi.siguiente() )
    if ( lpi.recuperar().equals(x) ) res.insertar(lpi.recuperar());
  return res;
}

public ListaConPI<V> recuperarValores(C c) {
  ListaConPI<V> res = new LEGListaConPI<V>();
  // Todas las entradas de clave c sólo pueden estar en la Lista Con PI
  // resultado de recuperarIguales de th
  ListaConPI<EntradaDic<C,V>> lpi =
    th.recuperarIguales(new EntradaDic<C,V>(c));
  for ( lpi.inicio(); !lpi.esFin(); lpi.siguiente() )
    res.insertar(lpi.recuperar().valor);
  return res;
}
```

Ejercicio 5.- Se dispone de una aplicación de radares de tráfico que permite llevar la contabilidad del número de veces que ha pasado un determinado coche por dicho radar superando el límite de velocidad. Para ello se consulta un *Diccionario* (*Diccionario<Matricula, Integer>*) implementado mediante una *Tabla Hash*. Se pide:

1. Sabiendo que el formato de una matrícula consta de una serie de 4 números seguido de 3 letras, completar la clase *Matricula* que figura a continuación. Añade los métodos y constructores que creas necesarios.

```
public class Matricula {
  private int numeros;
  private String letras;
  private String anyo;
  public Matricula(int numeros, String letras) {
    this.numeros = numeros;
    this.letras = letras;
  }
  public int numeros() { return numeros; }
  public String letras() { return letras; }
  public String anyo() { return anyo; }
  public String toString() { return numeros + " " + letras + " " + anyo; }
}
```

2. Con el fin de contabilizar el número de veces que ha pasado un coche se requiere actualizar el *Diccionario* de la aplicación: si la matrícula no está en el diccionario entonces el coche ha sido visto por primera vez. Si la matrícula ya estaba en el diccionario entonces hay que guardar en el diccionario que el coche se ha visto una vez más.

```
public static void registrarMatricula(Diccionario<Matricula,Integer> dM,
       Matricula m) {
  // Completar
}
```

Solución:

```
public class Matricula {
  // La clave de una Matricula es una combinación de sus números y sus letras
  private int numeros;
  private String letras;
  // El valor de una Matricula es el año de matriculación
  private String anyo;

  public Matricula (int numeros, String letras, String año) {
    this(numeros, letras);
    this.año = año;
  }

  public Matricula(int numeros, String letras) {
    this.numeros = numeros;
    this.letras = letras;
  }

  public boolean equals(Object x) {
    Matricula m = (Matricula) x;
    return this.numeros == m.numeros && this.letras.equals(m.letras);
  }

  public int hashCode() {
    return this.letras.hashCode() + numeros;
  }

  public int numeros() { return numeros; }
  public String letras() { return letras; }
  public String anyo() { return anyo; }
  public String toString() { return numeros + " " + letras + " " + anyo; }
}
```

```
public static void registrarMatricula(Diccionario<Matricula,Integer> dM,
       Matricula m) {
   int nMultas;
   try {
     nMultas = dM.recuperar(m).intValue();
   } catch (ElementoNoEncontrado e) {
     nMultas = 0;
   }
   dM.insertar(m, nMultas + 1);
}
```

Ejercicio 6.- Se dispone de una aplicación de control del número de veces que una dirección Web ha sido visitada. Para ello se utiliza un *Diccionario* (Diccionario<DireccionHTTP, Integer>) implementado mediante una Tabla Hash. Se pide completar la clase *DireccionHTTP* que figura a continuación:

```
public class DireccionHTTP {
    // Una DireccionHTTP, como por ejemplo http://www.host.com:80/datos/fichero.txt, consta de:
    // un servidor (www.host.com) , un puerto (80) y una dirección al recurso solicitado (/datos/fichero.txt)
    private String servidor;
    private int puerto;
    private String direccion;
    public DireccionHTTP(String servidor, int puerto, String direccion) {
       this.servidor=servidor;
       this.puerto=puerto;
       this.direccion = direccion;
    }
    public String getServidor() { return this.servidor; }
    public String getPuerto() { return this.puerto; }
    public String getDireccion() { return this.direccion; }
}
```

Solución:

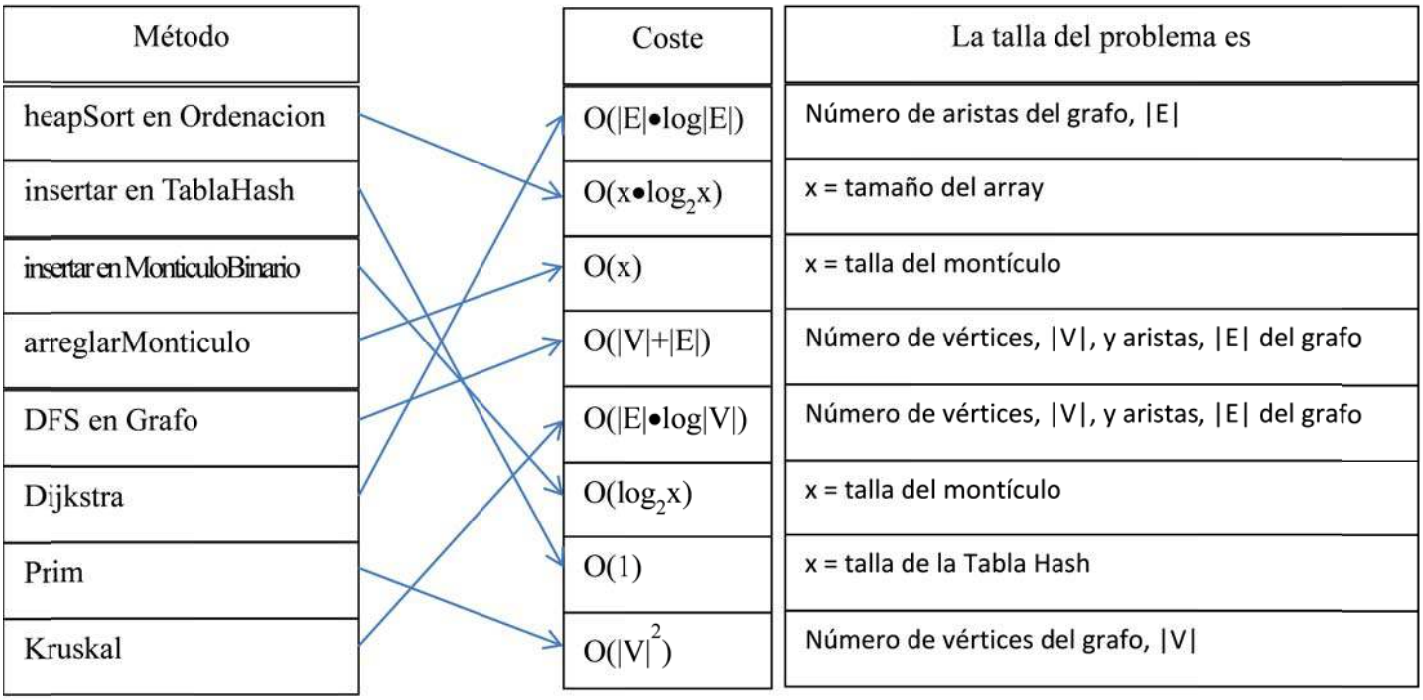
```
public class DireccionHTTP {
  private String servidor;
    private int puerto;
    private String direccion;
    public DireccionHTTP(String servidor, int puerto, String direccion) {
       this.servidor=servidor;
       this.puerto=puerto;
       this.direccion = direccion;
    }
    public String getServidor() { return this.servidor; }
    public String getPuerto() { return this.puerto; }
    public String getDireccion() { return this.direccion; }
    public boolean equals(Object o) {
       DireccionHTTP d = (DireccionHTTP) o;
       return this.servidor.equals(d.servidor) && this.puerto == d.puerto &&
              this.direccion.equals(d.direccion);
    }
    public int hashCode() {
       return (this.servidor + this.direccion + this.puerto).hashCode();
    }
}
```

Ejercicio 7.- Relacionar mediante flechas cada método con su coste y completar la descripción de la talla del problema:

Método
heapSort en Ordenacion
insertar en TablaHash
insertar en MonticuloBinario
arreglarMonticulo
DFS en Grafo
Dijkstra
Prim
Kruskal

Coste	La talla del problema es
$O(\|E\| \bullet \log \|E\|)$	
$O(x \bullet \log_2 x)$	
$O(x)$	
$O(\|V\|+\|E\|)$	
$O(\|E\| \bullet \log \|V\|)$	
$O(\log_2 x)$	
$O(1)$	
$O(\|V\|^2)$	

Solución:



Método	Coste	La talla del problema es
heapSort en Ordenacion	$O(\|E\| \bullet \log \|E\|)$	Número de aristas del grafo, |E|
insertar en TablaHash	$O(x \bullet \log_2 x)$	x = tamaño del array
insertar en MonticuloBinario	$O(x)$	x = talla del montículo
arreglarMonticulo	$O(\|V\|+\|E\|)$	Número de vértices, |V|, y aristas, |E| del grafo
DFS en Grafo	$O(\|E\| \bullet \log \|V\|)$	Número de vértices, |V|, y aristas, |E| del grafo
Dijkstra	$O(\log_2 x)$	x = talla del montículo
Prim	$O(1)$	x = talla de la Tabla Hash
Kruskal	$O(\|V\|^2)$	Número de vértices del grafo, |V|

Ejercicio 8.- Diseñar en la clase *GrafoD* un método recursivo, *caminoLongitudK*, que compruebe si existe un camino de longitud *k* entre dos vértices *u* y *v*.

```
public boolean caminoLongitudK(int u, int v, double k);
```

Nota: *se asume que no hay aristas con pesos negativos.*

Solución:

```
public boolean caminoLongitudK(int u, int v, double k) {
  if (k < 0) return false;
  if (k == 0) return u == v;
  ListaConPI<Adyacente> ady = elArray[u];
  for (ady.inicio(); !ady.esFin(); ady.siguiente()) {
     Adyacente aux = ady.recuperar();
     if (caminoLongitudK(aux.destino, v, k - aux.peso)) return true;
  }
  return false;
}
```

Ejercicio 9.- Se dice que un **Grafo Dirigido es Regular** si todos sus vértices tienen el mismo grado. Añadir a la clase *GrafoD* un método que compruebe si el grafo es regular con el menor coste temporal posible (visitando cada arista del grafo sólo una vez).

Solución:

```
public boolean esRegular() {
  int[] grado = gradosVertices();
  boolean esRegular = true;
  int grado0 = grado[0];
  for (int i = 0; i <= numVertices() && esRegular; i++ )
    esRegular = grado[i] == grado0;
  return esRegular;
}

// Devuelve un array con el grado de cada uno de los vértices
protected int[] gradosVertices () {
  int[] grado = new int[numVertices() + 1];
  for ( int i = 0; i <= numVertices(); i++ ) {
    ListaConPI<Adyacente> l = elArray[i];
    for ( l.inicio(); !l.esFin(); l.siguiente() ) {
      int codDes = l.recuperar().destino;
      grado[codDes]++;      // grado de entrada
      grado[i]++;           // grado de salida
    }
  }
  return grado;
}
```

Ejercicio 10.- Implementar en la clase *GrafoD* un método que permita averiguar si existe un camino simple entre dos vértices dados:

```
public boolean hayCaminoSimple(int vOrigen, int vDestino);
```

Solución:

```
public boolean hayCaminoSimple(int vOrigen, int vDestino) {
  visitados = new int[numVertices() + 1];
  return hayCamninoSimpleRec(vOrigen, vDestino);
}

protected boolean hayCaminoSimpleRec(int vOrigen, int vDestino) {
  if (vOrigen == vDestino) return true;
  visitados[vOrigen] = 1;
  ListaConPI<Adyacente> lpi = elArray[vOrigen];
  for (lpi.inicio(); !lpi.esFin(); lpi.siguiente()) {
    int vAdy = lpi.recuperar().destino;
    if (visitados[vAdy] == 0 && hayCaminoSimpleRec(vAdy, vDestino))
      return true;
  }
  return false;
}
```

Ejercicio 11.- Se quiere averiguar si desde un vértice dado en un Grafo Dirigido es posible alcanzar todos los demás vértices del mismo. Para ello, se pide diseñar en la clase *GrafoD* un método que devuelva el número de vértices que no son alcanzables desde un vértice origen dado.

```
public int numVerticesNoAlcanzables(int codOrigen);
```

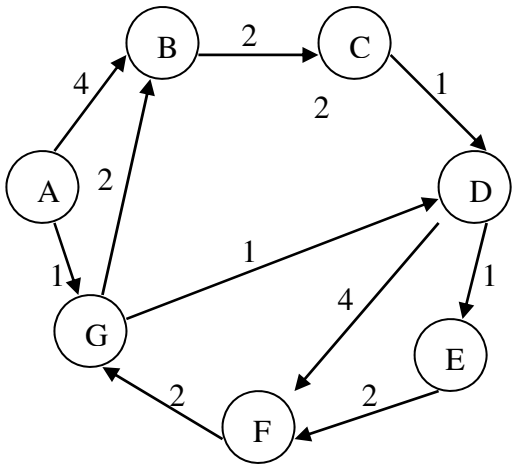
Solución:

```
public int numVerticesNoAlcanzables(int codOrigen) {
  visitados = new int[numVertices() + 1];
  recorridoProfundidad(codOrigen);
  int res = 0;
  for (int i = 1; i <= numVertices(); i++)
    if (visitados[i] == 0) res++;
  return res;
}

private void recorridoProfundidad(int origen) {
  visitados[origen] = 1;
  ListaConPI<Adyacente> l = adyacentesDe(origen);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    Adyacente a = l.recuperar();
    if (visitados[a.destino] == 0) recorridoProfundidad(a.destino);
  }
}
```

Ejercicio 12.- En un grupo de 7 amigos (Ana, Belen, Carlos, Damian, Esteban, Fernando y Gloria), Ana quiere quedar con Fernando pero lamentablemente no dispone de su número de móvil.
Para ello, Ana pretende hacerle llegar un mensaje a Fernando a través del resto de sus amigos. Modelamos este problema mediante un grafo dirigido, etiquetado y ponderado con aristas insertadas en el siguiente orden:

```
A B 4
B C 2
C D 1
D E 1
E F 2
F G 2
A G 1
G B 2
G D 1
D F 4
B D 2
```



Las ponderaciones reflejan las tarifas de las llamadas y las etiquetas correspondan con la primera letra del nombre del grupo de los amigos. Por ejemplo, una arista `A B 4` significa que Ana sabe el número de teléfono de Belén y mandarle un mensaje le cuesta 4 céntimos de euro.
Se pide:

1. Construir el Grafo correspondiente según su representación con Listas de Adyacencia.
2. Asumir que los vértices se numeran conforme aparecen en la secuencia de inserción de las aristas y que los adyacentes se van insertando al final de las listas de adyacencia.
3. Realizar una traza completa del algoritmo empleado (mostrando vértice, qPrior, distanciaMin y caminoMin).
4. ¿Cuál sería la secuencia de mensajes que se deberían realizar en el grupo para que el coste global de que Ana le haga llegar el mensaje a Fernando sea mínimo?

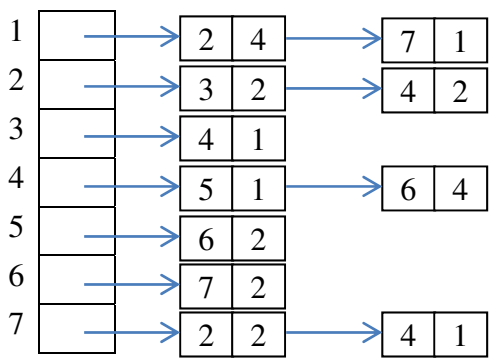
Solución:

Representación del grafo

Etiquetas:

1	2	3	4	5	6	7
Ana	Belen	Carlos	Damian	Esteban	Fernando	Gloria

Listas de adyacencia:



Traza de Dijkstra

	distanciaMin							caminoMin							
V	1	2	3	4	5	6	7	1	2	3	4	5	6	7	qPrior
	0	∞	∞	∞	∞	∞	∞	-1	-1	-1	-1	-1	-1	-1	(1,0)
1	0	4	∞	∞	∞	∞	1	-1	1	-1	-1	-1	-1	1	(7,1),(2,4)
7	0	3	∞	2	∞	∞	1	-1	7	-1	7	-1	-1	1	(2,4),(4,2),(2,3)
4	0	3	∞	2	3	6	1	-1	7	-1	7	4	4	1	(2,4),(2,3),(5,3),(6,6)
2	0	3	5	2	3	6	1	-1	7	2	7	4	4	1	(2,4),(5,3),(6,6),(3,5)
5	0	3	5	2	3	5	1	-1	7	2	7	4	5	1	(2,4),(6,6),(3,5),(6,5)
2	ya visitado							ya visitado							(6,6),(3,5),(6,5)
3	0	3	5	2	3	5	1	-1	7	2	7	4	5	1	(6,6),(6,5)
6	0	3	5	2	3	5	1	-1	7	2	7	4	5	1	(6,6)
6	ya visitado							ya visitado							∅

Camino mínimo:

<1, 7, 4, 5, 6> ⇒ <Ana, Gloria, Damian, Esteban, Fernando>
Coste total: 5 céntimos