Resolucion de la Recuperacion del Tercer Parcial junio 2014.pdf

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

Resolución de la Recuperación del Tercer Parcial de EDA (18 de Junio de 2014)

1.- En la clase `ABB` (ver Anexo), **se pide** implementar un método público `esABB` que, con el menor coste posible, compruebe que el Árbol Binario (AB) sobre el que se aplica es, en efecto, un ABB. **(2.5 puntos)**

La siguiente solución se basa en que el recorrido In-Orden de un ABB obtiene sus datos ordenados Asc.

```
public boolean esABB() {
  if (raiz == null) return true;
  E[] res = (E[]) new Comparable[talla];
  inOrden(res, raiz, 0);
  for (int i = 0; i < res.length - 1; i++)
      if (res[i].compareTo(res[i+1]) > 0) return false;
  return true;
}
protected int inOrden(E[] res, NodoABB<E> actual, int i) {
  if (actual == null) return i;
  i = inOrden(res, actual.izq, i);
  res[i++] = actual.dato;
  return inOrden(res, actual.der, i);
}
```

Alternativamente, aplicando la definición de ABB, se tendría el siguiente método:

```
public boolean esABB() {
  return esABB(this.raiz, null, null);
}
protected boolean esABB(NodoABB<E> actual, E min, E max) {
  if (actual == null) return true;
  if (min != null && actual.dato.compareTo(min) <= 0) return false;
  if (max != null && actual.dato.compareTo(max) >= 0) return false;
  return esABB(actual.izq, min, actual.dato) && esABB(actual.der, actual.dato, max);
}
```

Supón que el AB es completo; indica y justifica brevemente el coste temporal asintótico de tu método. **(0.5 puntos)**

Para ambas soluciones, talla = N, o número de nodos del AB.

Para la primera solución no hay instancias significativas, pues siempre se realiza el recorrido In-Orden del AB con coste lineal con su talla. Por tanto, $T(N) \in \Theta(N)$.

Para la segunda solución sí hay casos, pues se trata de una Búsqueda del primer elemento del AB que incumpla la propiedad del ABB. Así, …

-Mejor caso: el dato contenido en el hijo Izquierdo de la raíz del AB es mayor que el que ocupa la raíz (`max`). Por tanto, $T(N) \in \Omega(1)$.

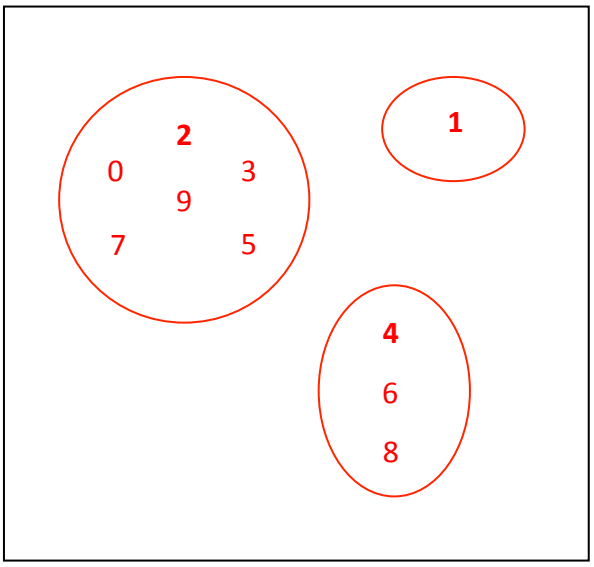
-Peor caso: la raíz del AB, i.e. el propio AB, es un ABB, por lo que para comprobarlo se realizan dos llamadas recursivas en secuencia en cada nodo `actual`. Por tanto, $T(N) \in O(N)$.

2- Dado el siguiente MF-Set:

2	-1	-4	2	-3	9	4	9	6	3

(2 puntos)

a) Dibujar en el siguiente recuadro cada uno de los subconjuntos disjuntos que contiene. **(0.4 puntos)**



b) Mostrar cómo se van transformando los árboles después de ejecutar cada una de las siguientes operaciones, haciendo uso de la unión por rango y la compresión de caminos. **(1.6 puntos)**
Nota: al unir dos árboles con la misma altura, el primer árbol deberá colgar del segundo.

merge(9,1)

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

find(5)

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

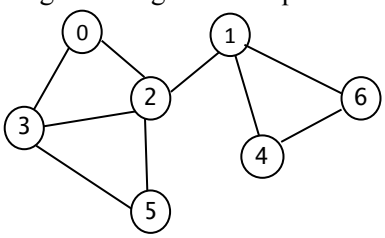
merge(4,1)

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

find(8)

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

3.- En teoría de grafos un puente es una arista tal que si fuese eliminada de un Grafo No Dirigido incrementaría el número de componentes conexas de éste. Nótese entonces que un Grafo Conexo dejaría de serlo si se elimina una arista puente, como por ejemplo en el Grafo de la siguiente figura en el que la arista `(1, 2)` es un puente.



En la clase `Grafo` (ver Anexo), **se pide** diseñar el método `esAristaPuente` que, con el menor coste posible, compruebe si una arista `(i, j)` de un Grafo es una arista puente. **(2.5 puntos)**

```
//SII es un Grafo No Dirigido
public boolean esAristaPuente(int i, int j) {
  visitados = new int[numVertices()];
  DFSSinIJ(i, i, j);
  if (visitados[j] == 0) return true; else return false;
}
protected void DFSSinIJ(int v, int i, int j) {
  visitados[v] = 1; ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 0)
       if (!(v == i & w == j)) //si (v, w) NO es la arista (i, j) que se busca ...
         if (w == j) visitados[w] = 1; //si se alcanza j, (i, j) NO puede ser un puente
         else        DFSSinIJ(w, i, j); //continua la búsqueda
  }
}
```

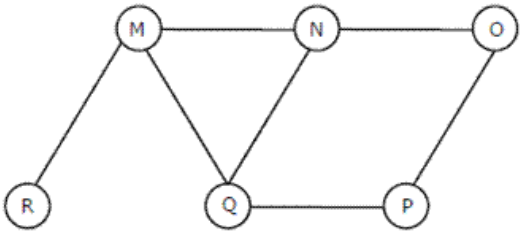
Una versión menos eficiente del DFS de un vértice, aunque también válida, sería:

```
protected void DFSSinIJ(int v, int i, int j)
  visitados[v] = 1; ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().getDestino();
    if (visitados[w] == 0 && !(v == i & w == j)) DFSSinIJ(w, i, j);
  }
}
```

4-. Se pide resolver las cuestiones sobre grafos que figuran a continuación: **(2.5 puntos)**

(a) Dado el Grafo de la figura, indicar cuál de los siguientes es el resultado de su Recorrido BFS y por qué.

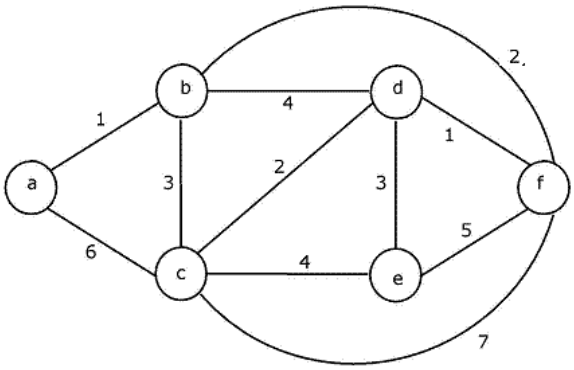
M N O P Q R
N Q M P O R
Q M N P R O
Q M N P O R



El resultado del BFS es `Q M N P R O` porque `M N` y `P` son adyacentes a `Q`, `R` es adyacente a `M` y `O` a `N`.

(b) Dado el Grafo de la figura, indicar cuál de las siguientes secuencias de aristas NO puede formar parte del Árbol de Recubrimiento Mínimo (ARM) que obtiene el algoritmo de Kruskal y por qué.

(a,b) (d,f) (b,f) (d,c) (d,e)
(a,b) (d,f) (d,c) (b,f) (d,e)
(d,f) (a,b) (d,c) (b,f) (d,e)
(d,f) (a,b) (b,f) (d,e) (d,c)



La secuencia que NO puede formar parte del ARM es la última porque la arista `(d,e)` no puede añadirse antes que una que, como la `(d,c)`, "pesa" menos: el algortimo de Kruskal siempre añade en cada paso la arista del Grafo que menos "pesa" y, además, no es una arista Hacia Atrás.

ANEXO

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

```
class NodoABB<E> { E dato; NodoABB<E> izq, der; NodoABB(E dato) {…} }
public class ABB<E extends Comparable<E>> {
       protected NodoAB<E> raiz; protected int talla;
       public ABB() {…}
       public boolean esVacio() {…}
       public int tamanyo() {…}
       …
       public String toStringPreOrden() {…}
       public String toStringInOrden() {…}
       public String toStringPostOrden() {…}
       public String toStringPorNiveles() {…}
}
```

Las clases `Grafo` y `Adyacente` del paquete `grafos`.

```
public abstract class Grafo {
    protected int visitados[]; // Para marcar los vértices visitados en un DFS o BFS
    protected int ordenVisita; // Para establecer el orden de visita de un vértice en un DFS o BFS
    protected Cola<Integer> q; // Para el Recorrido BFS sin pesos
    public abstract int numVertices();
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public String toString() {…}
    public int[] toArrayDFS() {…}
    protected int[] toArrayDFS(int i, int[] res) {…}
    public int[] toArrayBFS() {…}
    protected int[] toArrayBFS(int i, int[] res) {…}
    …
}
public class Adyacente {
    protected int destino; protected double peso;
    public Adyacente(int v, double peso) {…}
    public int getDestino() {…}
    public double getPeso() {…}
    public String toString() {…}
}
```