Grado en Ingeniería Informática

Estructura de Datos y Algoritmos, 2015/2016

Convocatoria Ordinaria (Evaluación Continua)

Nombre y Apellidos:

Grupo:

Instrucciones

1. Lee atentamente las preguntas, cerciórese de haber entendido lo que se pregunta e intente responder sólo a eso. Revise que la solución que va a proponer no incumple alguna condición del enunciado. Se valorará la concreción, la síntesis y la claridad.
2. Escriba su nombre y apellidos en este página y en todas las hojas de cuadros que utilice.
3. Cuando entregue su solución, por favor, enumere todas las páginas.
4. Use un bolígrafo. No se corregirá nada que esté escrito a lapicero.
5. No está permitido salir del aula por ningún motivo hasta la finalización del examen.
6. Desconecte su móvil durante el examen.
7. En la solución de este examen no se permite utilizar ninguna clase del API de Java tipo ArrayLink o LinkedList, tampoco arrays.

ENUNCIADO:

El Metro de Madrid es una red de ferrocarril metropolitano en constante cambio debido a la creación de nuevas estaciones y al cierre (temporal o permanente) de otras. En la actualidad, cuenta con un total de 301 estaciones, aunque como se ha dicho anteriormente esta cifra suele cambiar con frecuencia.

PROBLEMA 1 (1 punto) Para almacenar la lista de estaciones de metro sería posible utilizar un array (estructura estática) o bien una lista enlazada (estructura dinámica). Razona las respuestas a las siguientes preguntas:

a) ¿Cuál es la complejidad de añadir una nueva estación en un array?, ¿y en una lista enlazada? (0.3 puntos).

Respuesta:

En un array, el peor caso sería aquel en el que no hay espacio suficiente para guardar la nueva estación, siendo necesario ampliar el tamaño de dicho array. En este caso, sería necesario localizar espacio suficiente para almacenar la nueva versión del array ampliada, y mover entonces todos los elementos (estaciones) del antiguo array al nuevo, asegurándonos de crear la nueva estación en su

posición correspondiente.
En una lista simplemente enlazada, el peor de los casos sería aquel en el que debemos añadir la estación al final de la lista simplemente enlazada, y por tanto, sería necesario recorrer toda la lista. Por tanto, la complejidad sería lineal (O(n)).

b) Si conocemos la posición de la estación dentro del array o de la lista (es decir, su índice), ¿cuál es la complejidad de consultar una determinada estación en un array?, ¿y en una lista enlazada? (0.3 puntos).
Respuesta:
En un array, la complejidad de consultar un elemento en una determinada posición será constante (O(1)), ya que es posible acceder directamente al elemento a partir de la dirección de memoria del array + posición.
Sin embargo, en la lista enlazada, será necesario recorrer los nodos de la lista hasta encontrar el nodo de la posición buscada, que en el peor de los casos será el último nodo. Por tanto, la complejidad será lineal

c) ¿Qué estructura recomendarías si la mayoría de las operaciones que se van a realizar sobre dicha estructura van a ser de tipo lectura?, ¿y si son operaciones de actualización (inserción o borrado)? (0.4 puntos).
Respuesta:
Si la mayoría de las operaciones van a ser de tipo lectura lo más recomendable es que la estructura utilizada sea un array. Sin embargo, si las operación son de inserción o borrado, será más eficiente utilizar una estructura dinámica (lista enlazada) ya que no tendremos que preocuparnos por el tamaño de la lista.

PROBLEMA 2 (2 puntos).
Para simplificar el problema, vamos a suponer que de cada estación es necesario conocer únicamente su nombre y su estado (abierta o cerrada temporalmente). Aquí mostramos una implementación de clase Station aunque si lo necesitas, puedes modificarla incorporando nuevos atributos o métodos para resolver los diferentes apartados del problema.

```
public class Station {

    public String name;
    public char state;

    public Station(String n, char s) {
        name=n;
        state=s;
    }
}
```

Crea una clase Metro, que implemente una lista simplemente enlazada y que permita almacenar la lista de todas las estaciones. No tienes que desarrollar todos métodos de la lista simplemente enlazada, sólo será necesario que desarrolles el siguiente método:

- **boolean changeStatus(String nameStation, char state)**: el objetivo del método es cambiar el estado de una determinada estación o incluso borrar dicha estación de la lista, si lo que se solicita es su cierre permanente. En concreto, el método recibe el nombre de la estación a buscar en la lista y un argumento state de tipo char que va a indicarnos el tipo de operación que debe ser realizada:
  - Si state = 'o', en este caso, el método deberá buscar la estación y cambiar su estado a abierto ('o').

◦ Si state = 'c', en este caso, el método deberá buscar la estación y cambiar su estado a cerrado ('c').
◦ Si state = 'b', en este caso, no hay que modificar el estado de la estación, sino que debe ser eliminada de la lista de estaciones.
El método devuelve true si la operación se realizó correctamente, y false si la estación ya se encontraba en ese estado o no existe en la lista.

Nota 1: Si tu solución utiliza otros métodos de la interfaz IList (por ejemplo, isEmpty(), getSize(), getIndexOf(), removeAt(), etc), **deberás incluir su implementación**.
Nota 2: Las implementaciones basadas en arrays no serán evaluadas.

Solución:

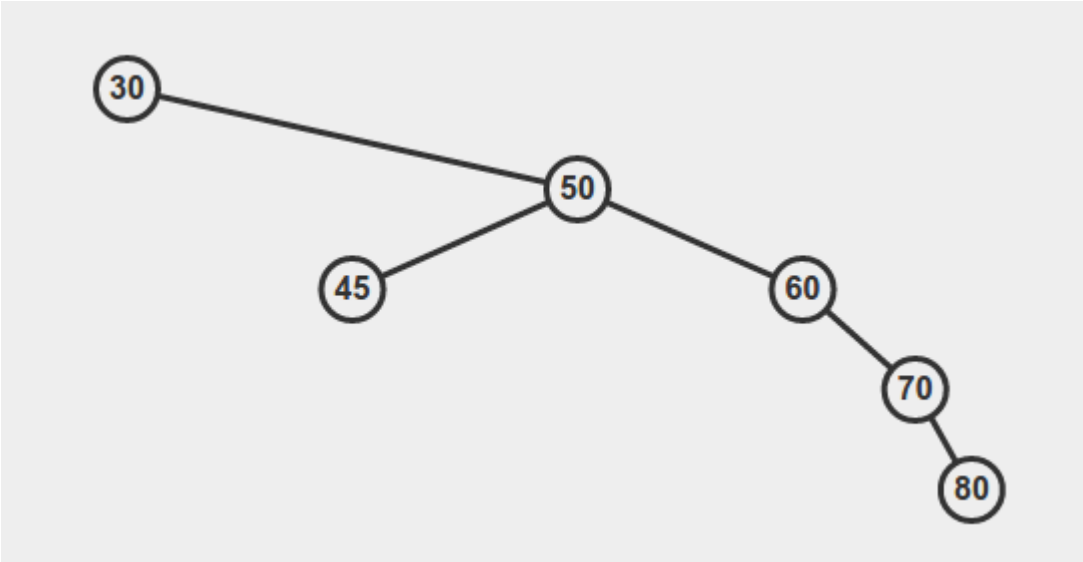
```
public boolean changeStatus(String nameStation, char state){
    boolean found=false;
    Station prev=null;
    for (Station it=firstStation;it!=null && !found;it=it.next) {
        if (nameStation.compareToIgnoreCase(it.name)==0) {
            if (state!='b') {
                if (state==it.state) {
                    return false;
                } else it.state=state;
            }
            else {
                //we have to remove this station
                if (prev==null)
                    //it is the first station
                    firstStation=firstStation.next;
                else prev.next=it.next;

            }
            found=true;
        }
        prev=it;
    }
    return found;
}
```

PROBLEMA 3 (2 puntos). Implementa un método que reciba un array de String y que lo ordene. El método debe implementar el algoritmo de **mergesort**.

Solución:

```
public static String[] sort(String[] s){
    if(s.length>1){
        int middle=s.length/2;
        String s1[]=new String[middle];
        String s2[]=new String[s.length-middle];

        for(int i=0;i<s1.length;i++)
            s1[i]=s[i];

        for(int i=0;i<s2.length;i++)
            s2[i]=s[i+s1.length];

        return merge(sort(s1),sort(s2));
    }

    else return s;
}

public static String[] merge(String[] s1, String[] s2){
    int i1=0,i2=0;
    int i=0;
    String[] mergeArray=new String[s1.length+s2.length];

    while(i1<s1.length && i2<s2.length){
        if(s1[i1].compareTo(s2[i2])<0) mergeArray[i++]=s1[i1++];
        else mergeArray[i++]=s2[i2++];
    }

    while(i1<s1.length) mergeArray[i++]=s1[i1++];
    while(i2<s2.length) mergeArray[i++]=s2[i2++];

    return mergeArray;
}
```

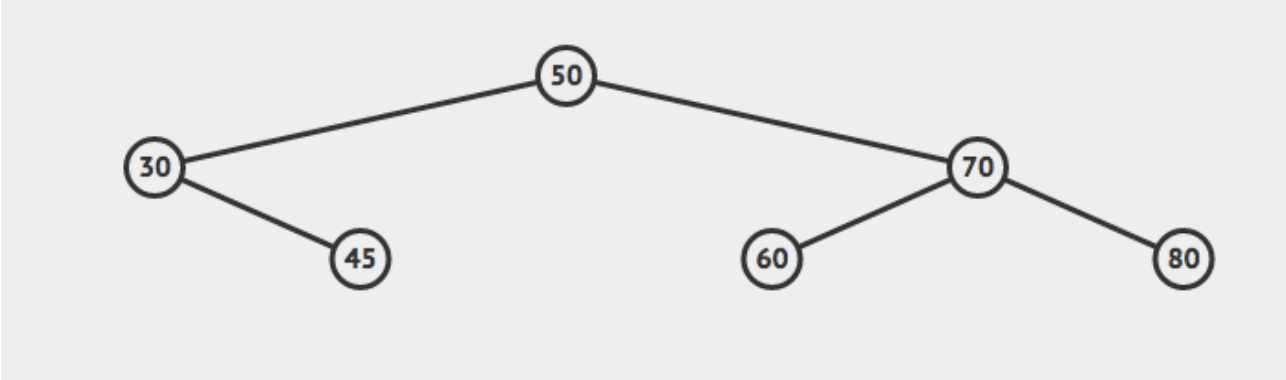
PROBLEMA 4 (1 punto). Dado el siguiente árbol binario de búsqueda:



a) (0.4 puntos) Dibuja el resultado tras aplicar los pasos necesarios para obtener el árbol perfectamente equilibrado equivalente (su equilibrado en tamaño).

b) (0.4 puntos) Dibuja el resultado tras aplicar los pasos necesarios para obtener el árbol AVL equivalente (equilibrado en altura).

Ambos equilibrios obtienen el mismo árbol.



d) (0.2 puntos) ¿Cuál es el objetivo final que se persigue con las operaciones de equilibrado?
Respuesta:
Mantener el árbol balanceado (en tamaño o altura) para que la complejidad de las operaciones se mantenga logarítmica.

PROBLEMA 5 (2.5 puntos).
Un elemento clave para el perfecto funcionamiento del Metro de Madrid es la correcta gestión de la plantilla de conductores. De cada conductor es necesario conocer su nombre y la línea a la que está asignado.
a) (0.5 puntos) ¿Qué ventajas ofrece almacenar el conjunto de conductores del Metro en una estructura de árbol binario de búsqueda frente a una estructura de lista enlazada?

Respuesta: En un árbol binario de búsqueda, la complejidad de las operaciones de búsqueda/borrado o inserción es logarítmica, frente a la complejidad lineal de las listas enlazadas.

Las siguientes imágenes muestran las clases Driver y DriverTree que permiten representar un conductor y un árbol binario de búsqueda para almacenar el conjunto de todos los conductores, respectivamente. La clase DriverTree está incompleta, ya que sólo incluye su método constructor y un método que permite la búsqueda de un determinado conductor a partir de su nombre.

```
public class Driver {

    String name;
    String line;

    Driver parent;
    Driver left;
    Driver right;

    public Driver(String n, String l) {
        name=l;
        line=l;
    }
}
```

```
public class DriverTree {

    Driver root;

    public DriverTree(Driver d) {
        root = d;
    }

    public Driver searchDriver(String name) {
        return searchDriver(root,name);
    }

    public Driver searchDriver(Driver itNode, String name) {
        if (itNode==null) {
            System.out.println(name + " is not a driver");
            return null;
        }

        if (name.compareTo(itNode.name)<0) {
            return searchDriver(itNode.left, name);
        } else if (name.compareTo(itNode.name)>0) {
            return searchDriver(itNode.right, name);
        } else {
            System.out.println(name + " founded!!!");
            return itNode;
        }
    }
```

b) (2 puntos) Implementa un método **void addNewDriver(String name, String line)** que permita añadir un nuevo conductor al árbol binario de búsqueda que almacena los conductores.

```
public void addNewDriver(String name, String line) {
    Driver newDriver=new Driver(name,line);
    if (root==null) root=newDriver;
    else addNewDriver(root,newDriver);

}

public void addNewDriver(Driver itNode, Driver newDriver) {
    if (itNode.name.compareTo(newDriver.name)==0) {
        System.out.println(newDriver.name + " already exists!!!");
        return;
    }

    if (newDriver.name.compareTo(itNode.name)<0) {
        if (itNode.left!=null)
            addNewDriver(itNode.left,newDriver);
        else {
            itNode.left=newDriver;
            newDriver.parent=itNode;
        }

    } else {
        if (itNode.right!=null)
            addNewDriver(itNode.right,newDriver);
        else {
            itNode.right=newDriver;
            newDriver.parent=itNode;
        }
    }

}
```

PROBLEMA 6 (1.5 puntos).

La red Metro puede ser implementada con un Grafo, donde las estaciones son los nodos del grafo y las aristas serían las conexiones entre las estaciones. Para simplificar el problema, las estaciones en lugar de ser representadas por su nombre, son identificadas con un índice. La siguiente imagen muestra la implementación (incompleta) de una red de estaciones basada en una matriz de adyacencias. Dicha implementación contiene algunos métodos como addNewStation (que añade una nueva estación), addConnection (que añade una conexión entre un par de estaciones), isConnection (método que comprueba si existe una conexión directa entre dos estaciones) y removeConnection (método que elimina la conexión entre dos estaciones).

Se pide:
a) (0.75 puntos) Implementa un método que calcule el número total de conexiones en la red de Metro.

b) (0.5 puntos) Implementa un método que reciba un índice (representa una estación) y muestre por pantalla los índices de las estaciones que tienen conexión directa con ella.
c) (0.25 puntos) Durante el curso hemos estudiado dos tipos de implementaciones para grafos: basada en matriz de adyacencias y basada en listas de adyacencias. ¿Qué ventajas ofrece una implementación de grafos basada en lista de adyacencias frente a la implementación basada en matriz de adyacencia?

```
public class MetroGraph {
    //maximum of 500 stations
    boolean connections[][]=new boolean[500][500];
    //current number of stations
    int numStations=0;
    //add a new station
    public void addNewStation() {
        if (numStations+1>=500) {
            System.out.println("We cannot add anymore stations");
            return;
        }
        numStations++;
    }
    //stations are represented by index rather by names
    //adds a connection between the stations i and j
    public void addConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            connections[i][j]=true;
            connections[j][i]=true;
        }
    }
    //checks if there is a connection between the stations i and j
    public boolean isConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            return connections[i][j];
        } else return false;
    }
    //remove the connection between the stations i and j
    public void removeConnection(int i, int j) {
        if (0<=i && i <numStations && 0<=j && j <numStations) {
            connections[i][j]=false;
        }
    }
```

Solución:
a)
En general, el metro debería ser representado como un grafo no dirigido porque las estaciones están conectadas en ambos sentidos. En este caso, la solución sería:

```
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
            for (int j=0;j<numStations;j++) {
                if (connections[i][j]) total++;
        }
    }
    total=total/2;
```

Una solución equivalente más eficiente podría ser:

```
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
        for (int j=i;j<numStations;j++) {
            if (connections[i][j]) total++;
        }
    }
    return total;
}
```

Al ser un grafo no dirigido, su matriz es simétrica y por tanto sólo es necesario revisar una parte de la matriz (a partir de la diagonal).

Por otro lado, el método remove nos puede hacer pensar que el grafo no es dirigido ya que sólo estamos borrando la conexión. Este método tiene sentido ya que podemos estar interesados en cerrar la conexión de una estación a otra pero en un solo sentido. La única diferencia, es que en este caso, el grafo sería un grafo no dirigido. Para los alumnos que hayan considerado esta opción, la solución sería:

```
public int numTotalConnections() {
    int total=0;
    for (int i=0;i<numStations;i++) {
        for (int j=0;j<numStations;j++) {
            if (connections[i][j]) total++;
        }
    }
    return total;
}
```

b)

```
public void showStationsConnectedTo(int i) {
    for (int row=0;i<numStations;row++) {
        if (connections[row][i]) System.out.println(row);
    }
}
```

c) La implementación basada en listas requiere menos espacio ya que sólo se almacenan la información sobre las aristas existentes. Además algunas operaciones (por ejemplo, recorrido todas las aristas) también son computacionalmente más costosas en la implementación basada en matriz que en listas.