Grado en Ingeniería Informática

Estructura de Datos y Algoritmos, 2015/2016

Convocatoria Extraordinaria

Nombre y Apellidos:

Grupo:

Instrucciones

1. Lee atentamente las preguntas, cerciórate de haber entendido lo que se pregunta e intenta responder sólo a eso. Revisa que la solución que vas a proponer no incumple alguna condición del enunciado. Se valorará la concreción, la síntesis y la claridad.
2. Escribe tu nombre y apellidos en esta página y en todas las hojas de cuadros que utilices.
3. Cuando entregues tu solución, por favor, enumera todas las páginas.
4. Usa 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. Desconecta tu 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, salvo que un enunciado especifique lo contrario.

ENUNCIADO:

Problema 1 (2.5 puntos) En este problema queremos usar TADs lineales para detectar y borrar una sucesión de números enteros donde el siguiente elemento es el doble del anterior (ver ejemplo). Asumamos que tenemos un TAD lineal implementado en una lista simplemente enlazada que almacena enteros (tipo "int" de JAVA). Se pide implementar un método que busque dentro del TAD la sucesión descrita anteriormente y, en caso de existir, la elimine, dejando únicamente el primer elemento de la sucesión. Además, el método debe devolver el número total de nodos eliminados. El método no recibe parámetros.

El siguiente ejemplo muestra el resultado tras la ejecución del método:

Inicio:
*Contenido del TAD inicialmente (un ejemplo): 4, 1, 2, 4, 8, 9, 3, 2, **5, 10, 20**, 2, 1, 7*
Contenido de la lista una vez ejecutado el método: 4,1,9,3,2,5,2,1,7
Número total de nodos eliminados: 5

Especifica el caso peor y la complejidad (orden) del método implementado. Haz lo mismo con el caso mejor.
Nota: Cualquier método auxiliar que se quiera usar deberá ser implementado a continuación del que se pide.

Solución:

```
public int removeSequence(){
    SNode prev=firstNode;
    int count=0;
    if(prev!=null){
        int ref;
        ref=prev.elem;
        if(prev.next!=null){
            for (SNode nodeIt = prev.next; nodeIt != null; nodeIt = nodeIt.next) {
                if(nodeIt.elem==2*ref){
                    ref*=2;
                    prev.next=nodeIt.next;
                    count++;
                }
                else{
                    ref=nodeIt.elem;
                    prev=nodeIt;
                }
            }
        }
    }
    return count;
}
```

El caso mejor requiere que la lista esté vacía, en cuyo caso la complejidad es O(1).
El caso peor y el caso promedio es O(n) porque siempre es necesario recorrer toda la lista.

Problema 2 (2 puntos): Considera el siguiente método iterativo que comprueba si un número es primo. Recuerda que un número es primo si es mayor que cero y solo es divisible por 1 y por él mismo.

```
public boolean isPrimeIt(int number) {
        if (number <= 0) {
                System.out.println("The number must be greater than 0.");
                return false;
        } else {
                boolean prime= true;
                for (int x=2; x<number && prime; x++) {
                    if (number % x == 0)  prime= false;
                }
                return prime;
        }
}
```

1. (0.5 puntos) Calcula la función tiempo de ejecución y la función orden de complejidad del método isPrimeIt().
2. (1.5 puntos) Implementa un método **recursivo public boolean** isPrime que compruebe si un número es primo.

Solución:

a)
```
public static boolean isPrimeIt(int number) {
        if (number <= 0) { //1
                System.out.println("The number must be greater than 0."); //1
                return false; //1
        } else {
                for (int div=2; div<number; div++) // 1+ (n-2)+1 + (n-2) = 2n-2
                  if (number % div == 0)  //2(n-2)=2n-4
                          return false;                 //1
                return true; //1
        }
}
//T(n)= 1+ 2n-2 +2 +2n-4+1 = 4n-2 y O(n) => Complejidad lineal
```

b)
```
public static boolean isPrime(int number) {
        return isPrime(number, 2);
}

public static boolean isPrime(int number, int div) {
        if (number <= 0) {
                System.out.println("The number must be greater than 0.");
                return false;
        } else
                if (number==1 || number==div) //number == 1 o he llegado al número
                        return true;
                else
                        if (number % div == 0)
                                return false;
                        else
                                return isPrime(number, div+1);    //pruebo si el siguiente
entero es un divisor
        }
```

Problema 3 (3 puntos): Dada la clase que representa un nodo binario:

```
public class BSTNode {
    public int elem;

    public BSTNode parent;
    public BSTNode left;
    public BSTNode right;

    public BSTNode(int e) {
            elem=e;

    }
```

```
        //returns its rightmost node of its left child
        public  BSTNode predecessor(BSTNode node) {
                if (node==null) return null;
                BSTNode pred=node.left;

                if (pred!=null) {
                        while (pred.right!=null) {
                                pred=pred.right;
                        }
                }
                return pred;
        }

        //returns its leftmost node of its right child
        public  BSTNode successor(BSTNode node) {
                if (node==null) return null;
                BSTNode suc=node.right;
                if (suc!=null) {
                        while (suc.left!=null) {
                        suc=suc.left;
                        }
                }
                return suc;
        }

        //returns the size of this subtree
        public  int getSize(BSTNode node) {
                if (node==null) return 0;//
                return 1 + getSize(node.left)+getSize(node.right);
        }

}
```

a) (2 puntos) Implementa un método recursivo que reciba un objeto de tipo BSTNode y compruebe si satisface la definición de árbol binario de búsqueda (ABB). Los métodos *successor* y *predecessor* te pueden ayudar a diseñar la solución. Como ayuda te damos los casos base del método recursivo:

```
public  boolean isBST(BSTNode node) {
                //primer caso base: el nodo es null
                if (node==null) return true;
                //segundo caso base: el nodo es una hoja
                if (node.left==null && node.right==null) return true;

                BSTNode pred=predecessor(node);
```

```
        BSTNode suc=successor(node);

        //Completa el código


    }
```

Solución:

```
public  boolean isBST(BSTNode node) {
        if (node==null) return true;
        else if (node.left==null && node.right==null) return true;

        BSTNode pred=predecessor(node);
        BSTNode suc=successor(node);

        if (pred!=null && suc!=null)
            return pred.elem<node.elem && suc.elem >node.elem
                        && isBST(node.left) && isBST(node.right);
        else if (suc!=null)
            return suc.elem>node.elem && isBST(node.right);
        else //(pred!=null)
            return pred.elem<node.elem && isBST(node.left);
    }
```

b) (0.5 puntos) Implementa un método que calcule el factor de equilibrio por tamaño de un determinado nodo.

Solución:

```
private int balanceFactorSize(BSTNode node) {
    if (node==null) {
        return 0;
    }
    return Math.abs(getSize(node.right)-getSize(node.left));
}
```

c) (0.5 puntos) Implementa el método que compruebe si el árbol está perfectamente equilibrado (todos sus nodos tienen factor de equilibrio en tamaño menor o igual que 1).

Solución:

```
public boolean isBalanceSize () {
        return isBalanceSize(root);
}

public boolean isBalanceSize(BSTNodeContact node) {
        if (node==null) return true;
      return (balanceFactorSize(node)<=1 && isBalanceSize(node.left) &&  isBalanceSize(node.right));
}
```

Problema 4 (2.5 puntos):

La Red Social Mynet quiere almacenar información sobre los usuarios y las relaciones entre ellos. Para este propósito, se utiliza una estructura de grafos. Hay cuatro tipos de relaciones: las relaciones familiares de primer grado (para los padres, hermanos y hermanas), las relaciones familiares de otro grado (para los tíos, primos, abuelos, etc.), las relaciones de trabajo y las relaciones de amistad. No puede haber personas que comparten dos tipos de relaciones al mismo tiempo (es decir, si por

ejemplo alguien está vinculado a mí a través de una relación de trabajo, esa persona no puede estar vinculado a mí a través de un relación de amistad). En otras palabras, cada par de personas sólo podrá mantener un único tipo de relación.

El siguiente código se utiliza para definir la estructura:

```
public class GraphMA {

        boolean matrix[][];
        //maximum number of vertices
        int maxVertices;
        //current number of vertices
        int numVertices;
        //true if the graph is directed, false in other case
        boolean directed;

        //constructor
        public GraphMA(int max, boolean d) {
                matrix=new boolean[max][max];
                maxVertices=max;
                numVertices=0;
                directed=d;
        }
}
```

1. ¿Es esta implementación correcta para representar la red social descrita en el enunciado? Si la respuesta es no, explica por qué y da una implementación correcta. (0.75 puntos)

   Solución:

   No es correcta porque no permite representar el tipo de relación.

```
public class GraphMA {

        public enum relationType {FAM_FIRST_DEGREE,
                        FAM_SECOND_DEGREE, WORK, FRIEND};

String matrix[][];
//maximum number of vertices
int maxVertices;
//current number of vertices
int numVertices;
//true if the graph is directed, false eoc
boolean directed;

//constructor
public GraphMA(int max, boolean d) {
        matrix=new String[max][max];
        maxVertices=max;
        numVertices=0;
        directed=d;
        }
}
```

2. (0.75 puntos) ¿La red social debería representarse como un grafo dirigido o no? Implementa también un método que permita relacionar dos usuarios con un determinado tipo de relación.

Solución

La interpretación común sería que la red social se corresponde con un grafo no dirigido. Aunque también se podría interpretar que las relaciones no tienen por qué ser simétricas. Así, por ejemplo, tu puedes considerar a alguien como un amigo, pero esa persona sin embargo te considera como un compañero de trabajo.

```
public void addEdge(int i, int j, relationType rel) {
        if (checkIndex(i) && checkIndex(j)) {
                matrix[i][j]=rel.toString();
                if (!directed)          matrix[j][i]=rel.toString();
        }
}

public boolean checkIndex(int i){
        if (i<0 || i>=numVertices) return false;
        return true;
}
```

3. Implementa el método show() que muestra la red social, es decir, las relaciones entre los usuarios. ¿Cuál es la complejidad de este método?, ¿su implementación basada en listas de adyacencia permitiría reducir la complejidad? (1 punto)

Solución

```
public void show(){
        if (numVertices==0) {
                System.out.println("The graph is empty!!");
                return;
        }
        for (int i=0;i<numVertices;i++) {
                for (int j=0;j<numVertices;j++) {
                        System.out.print(matrix[i][j]+"\t");
                }
                System.out.println();
        }
}
```

$T(N) = 1 + (1 + n + n + 1) + n*(1 + n + n + 1) + n*n = 2n + 3 + 2n^2 + 2n + n^2 = 3n^2 + 4n + 3$

Por tanto, su complejidad es $O(n^2)$.

La implementación basada en la lista de adyacencias no reduce la complejidad ya que por un lado hay que recorrer el array de vértices y para cada vértice recorrer su lista de vértices adyacentes. Por tanto, su complejidad seguiría siendo $O(n^2)$.