Estructura de Datos y Algoritmos. Ingeniería Informática Universidad Carlos III de Madrid EXAMEN FINAL. CONVOCATORIA ORDINARIA. CURSO 2017-2018	uc3m | Universidad Carlos III de Madrid

Nombre y Apellido: **Grupo:**

Problema 1 (2 puntos) – Supongamos que BSTree es una clase que implementa un árbol de búsqueda binario, con todos sus métodos implementados (¡no tienes que implementarlos!).

```
public class BSTree {
        BSNode root;
        …
        public void insert(int key) {
        …
        }
        …
}
```

Escribe un método estático que tome como parámetro un array de enteros ordenado de manera ascendente (no es necesario comprobar que el array esté ordenado) y que devuelva un árbol BST perfectamente equilibrado.

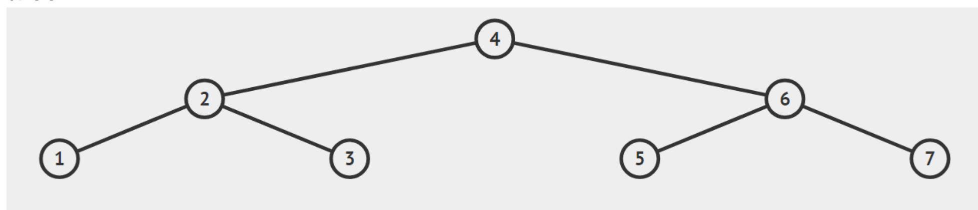
AYUDA:

- No es necesario obtener el factor de equilibrio de un nodo.
- No es necesario aplicar ninguna rotación ni mover nodos de un subárbol a otro.
- Piensa cómo aprovechar la suposición de que el array ya está ordenado.
- Debes utilizar el enfoque de divide y vencerás.
- Puedes utilizar el método de inserción de BSTree.
- Puedes escribir un método auxiliar que tome más parámetros de entrada si lo necesitas.

Ejemplos:

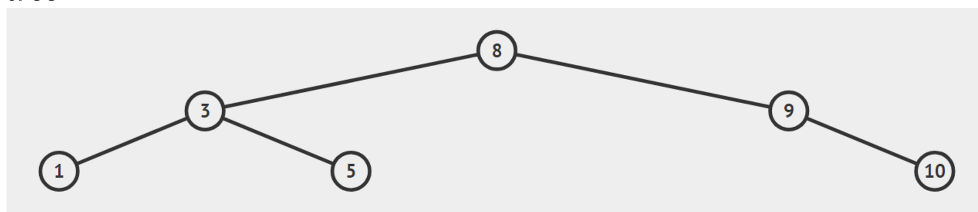
A = [1, 2, 3, 4, 5, 6, 7]

tree =



A = [1,3,5,8,9,10]

tree =



Solución:

```
public static BSTree arrayToBST(int[] data) {
	BSTree tree=new BSTree();
	if (data!=null && data.lenght>0)
		arrayToBST (data, 0, data.length – 1, tree);
	return tree;
}
private static void arrayToBST(int[] data, int start,     int end, 
BSTree tree) {
		if (start > end) return;
		int mid = (end + start) / 2;
		tree.insert(data[mid]);
		arrayToBST(data, start, pos – 1, tree);
		arrayToBST(data, pos + 1, end, tree);
}
```

Problema 2 (1 puntos) –

A) (0.5) Supongamos que BSNode es una clase que implementa un nodo de búsqueda binario. Escribe un método no estático, ***equals***, que tome un objeto BSNode como parámetro de entrada y compruebe si es igual al objeto invocador. El método devuelve verdadero si son iguales y, en caso contrario, falso.

Consideramos que dos nodos son iguales cuando sus atributos tienen el mismo valor y sus padres y sus subárboles izquierdo y derecho también son iguales.

```
public class BSNode {
	Integer key;
	String elem;
	BSNode parent;
	BSNode left;
	BSNode right;

public boolean equals(BSNode obj) {
….
}
}
```

B) (0.25) Supongamos que BSTree es una clase que implementa un árbol de búsqueda binario. Escribe un método no estático, **equals**, que tome un objeto BSTree como parámetro de entrada, y compruebe si este árbol es igual al árbol invocador. Debes utilizar el método equals de BSNode como método auxiliar para realizar la comprobación.

```
public boolean equals(BSTree tree) {
…
}
```

C) (0.25) ¿Cuál es la complejidad temporal de los métodos (A y B) en la notación Big-O ?. Justifícalo.

Solución:

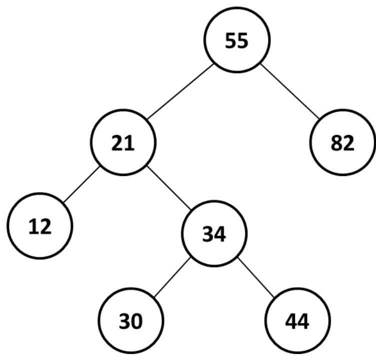
```
(a)
public boolean equals(BSTNode node){
	return equals(this,node);
}

public static boolean equals(BSTNode n1, BSTNode n2){
	 if (n1 == null && n2 == null) return true;

	 if (n1 != null && n2 != null) {
		return (n1.key == n2.key &&
	   n1.elem.compareTo(n2.elem)==0
			&& equals(n1.left, n2.left)
			&& equals(n1.right, n2.right));

	}
	//otherwise
	return false;
}
```

Note: compareTo te permite comparar objetos nulos. El método equals lanzaría una excepción cuando una de las dos cadenas fuese nula.

```
(b)
public boolean equals(BSTree tree){
	return equals(this,tree);
}

public static boolean equals(BSTree t1, BSTree t2){
	 if (t1 == null && t2 == null) return true;

	 if (t1 != null && t2 != null) {
	   if (t1.root!= null) return t1.root.equals(t2.root);
	   if (t2.root!= null) return t2.root.equals(t1.root);
	}

	//otherwise
	return false;
}
```

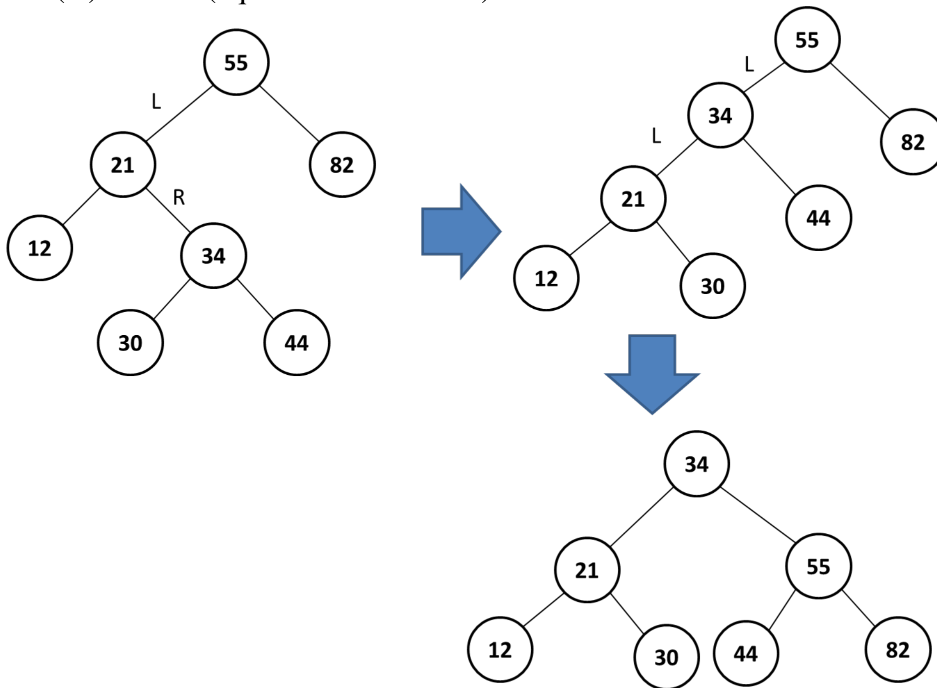
(c) Supón que n es el tamaño del subárbol que cuelga del nodo invocante (o del árbol invocante). Ambos métodos tienen complejidad linear, porque tienen que recorrer todos los nodes que cuelgan del nodo invocante o de la raíz del árbol invocante.

Problema 3 (1 punto):
A) (0.5) Equilibra (paso a paso) el siguiente árbol de búsqueda binaria para que satisfaga la propiedad AVL (equilibrio de altura).
B) (0.5) Además, equilíbralo (paso a paso) para que sea un árbol perfectamente equilibrado (equilibrio en tamaño).



Solución:

(A) – AVL (equilibrado en altura)



(B) – Equilibrado perfecto (equilibrado en tamaño).
El árbol resultante es el mismo que en el apartado anterior.

Problema 4 (2 puntos).

La universidad desea gestionar las asignaturas que son requisito previo para cursar otras asignaturas. De esta manera, cuando un alumno acude a matricularse, es posible comprobar que ha cursado todas las asignaturas necesarias. Así, si las asignaturas "Cálculo" y "Programación" son requisitos para "Estructura de Datos y Algoritmos (EDA)" implicará que no es posible cursar EDA sin haber cursado previamente "Cálculo" y "Programación".

Suponiendo que como máximo, para unos estudios de grado determinado hay un máximo de 20 asignaturas.

1) (0.25) ¿Qué estructura(s) de datos es la más adecuada para representar las relaciones entre las asignaturas y los pre-requisitos?. Explícalo. ¿Esas relaciones son simétricas?
2) (0.25) Para implementar dicha estructura, escribe las cabeceras de las clases, los atributos y un método constructor que toma un array de los nombres de las asignaturas como parámetro de entrada. En ese constructor, no hace falta definir ninguna relación.
3) (0.5) Escribe un método no estático, **requiredFor**, que tome el nombre de una asignatura como parámetro de entrada, y devuelva una lista de todas las asignaturas que tienen como requisito previo la asignatura de entrada. Supongamos que las relaciones entre las asignaturas ya han sido almacenadas en la estructura elegida. Por ejemplo, EDA es un requisito previo para las asignaturas "Heurística y optimización", "Programación orientada a objetos", "Ingeniería del conocimiento", "Diseño de sistemas operativos", "Lenguajes formales y teoría de autómatas".
4) (0.5) Escribe un método no estático, **getRequiredSubjectsFor**, que tome el nombre de una asignatura como parámetro de entrada, y devuelva la lista de asignaturas que un alumno debería haber aprobado para poder matricularse en la asignatura de entrada. Por ejemplo, para matricularse en EDA, el alumno debería haber aprobado las asignaturas "Cálculo" y "Programación".
5) (0.5) Escribe un método no estático, **notAllowed**, que tome una lista con todas las asignaturas aprobadas por un alumno, y devuelva una lista con todas las asignaturas en las que el alumno aún no puede matricularse.
   Nota: En este problema, puedes utilizar clases de Java tales como LinkedList <String> o ArrayList<String> para implementar su solución.

Solución:

a) Un grafo es una estructura de datos adecuada para representar las asignaturas y las relaciones entre las asignaturas. El grafo sería no ponderado (las relaciones no tienen ningún peso) y dirigido (ya que las relaciones no son simétricas). Respecto a la implementación, como el número máximo de asignaturas es 20, usar una matriz puede ser una buena opción, porque usaría sólo 400 posiciones de memoria para almacenar las posibles relaciones. Sería una buena opción si el grafo es denso, es decir, si existen muchas relaciones entre las asignaturas. Por otro lado, el uso de una lista de adyacencias también sería una opción óptima tanto en complejidad espacial (sólo usas la memoria necesaria) y temporal (las mayorías de las operaciones tendrían complejidad linear).

(b)

```java
public class RequirementGraph  {
	public String[] subjects;
	public int num;
	ArrayList<Integer>[] lstAdjacents;

	public RequirementGraph(String[] subjects) {
			this.subjects=subjects;
			this.num=subjects.length;
			//we must initialize each adjacent list
			lstAdjacents=new ArrayList[num];
			for (int i=0; i<num; i++) {
				lstAdjacents[i]=new ArrayList();
			}
	}
```

(c)

```java
//returns all possible subjects that have as requirement the input subject
	public ArrayList<String> allows(String subject) {
		int index=getIndex(subject);
		if (index==-1) return null;
		ArrayList<String> result=new ArrayList<String>();
		for (int i=0; i<lstAdjacents[index].size();i++) {
			int adj=lstAdjacents[index].get(i);
			result.add(getNameSubject(adj));
		}
		return result;
	}
```

Note: You can suppose that the methods **getIndex** and **getNameSubject**.

(d)

```java
public ArrayList<String> getRequiredSubjectsFor(String subject) {
	int index=getIndex(subject);
	if (index==-1) return null;

	ArrayList<String> result=new ArrayList<String>();
	for (int i=0; i<num;i++) {
		for (int j=0; j<lstAdjacents[i].size();j++) {
			int adj=lstAdjacents[i].get(j);
			if (index==adj) result.add(getNameSubject(i));
		}
	}
	return result;
}
```

(e)

```
public ArrayList<String> notAllowed(ArrayList<String> lst) {
	if (lst==null) return null;
	ArrayList<String> notAllow=new ArrayList<String>();
	//traverse all subjects
	for (int i=0; i<subjects.length; i++) {
		String subject=subjects[i];
		//if the subject is already in the list, we do nothing
		//otherwise:
		if (!lst.contains(subject)) {
			//this subject has not been passed yet
			//we gets all required subject for this
			ArrayList<String>  req=getRequiredSubjectsFor(subject);
			boolean allowed=true;
			for (int j=0; j<req.size()&&allowed;j++) {
				String name=required.get(i);
				//if some of the required subjects is not in the list,
				//then this subject is not allowed. We must break the loop
				if (!lst.contains(name)) allowed=false;
			}
			if (!allowed) notAllow.add(subject);
			}
		}
	return notAllow;
}
```