Universidad Carlos III de Madrid

Grado en Ingeniería Informática
Departamento de Informática

Estructura de Datos y Algoritmos

Manual de Ejercicios

Authors: Harith Al-Jumaily
Julian Moreno Schneider
Juan Perea
Isabel Segura Bedmar

Junio 2011

Índice:

Complejidad 4

Listas, Pilas y Colas 10

Árboles 38

Caso Práctico 70

Capitulo 1

Complejidad

1.1

Escribe una clase en Java que realice las siguientes operaciones:

a) Obtener el máximo y el mínimo número de un array.
b) Ordenar los elementos de un array.
   Analiza la complejidad de ambas operaciones.
   En el caso de la ordenación de elementos de un array, ¿puedes proponer distintas soluciones?, ¿que solución es la mejor?

Utiliza Excel para dibujar los resultados.

Solución:

a)

```
public static int getMaxArray(int A[]) {
	if (A.length==0) {
		System.out.println("Array vacio");
		return -1;
	}
	int max=A[0];
	for (int i=1; i<A.length;i++) {
		if (A[i]>max) max=A[i];
	}
	return max;
}

Complejidad getMaxArray es lineal (O(n)).

public static int getMinArray(int A[]) {
	if (A.length==0) {
		System.out.println("Array vacio");
		return -1;
	}
	int min=A[0];
	for (int i=1; i<A.length;i++) {
		if (A[i]<min) min=A[i];
	}
	return min;
```

}

Complejidad getMinArray también es lineal (O(n)). Tiene que recorrer todo los elementos del array al menos una vez, por tanto, necesitará al menos n unidades de tiempo siendo n el tamaño del array.
Podemos prescindir de las costantes en el cálculo de la complejidad.

b) Ordenar los elementos de un array.

Existen varios algoritmos de ordenación (básicos: burbuja $O(n^2)$, insercción directa $O(n^2)$ y selección directa $O(n^2)$) (avanzados:quicksort y heapsort).

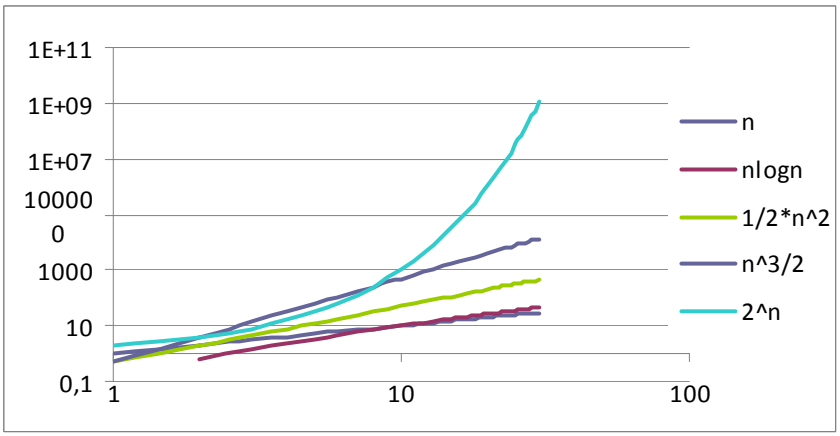
1.2

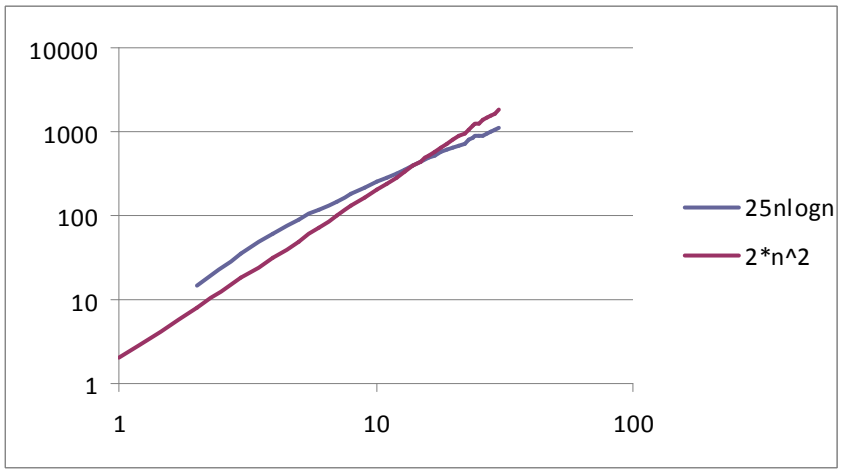
(a) Dibuja en escala log-log las siguientes funciones: $f(n) = n$, $g(n) = n \cdot \log(n)$, $h(n) = 1/2.n^2$, $j(n) = n^3/2$, $k(n) = 2^n$. La escala logarítmica utiliza tanto en el eje x como en el eje y una escala logarítmica.

(b) El algoritmo A usa $25n \cdot \log(n)$ operaciones mientras que el algoritmo B usa $2n^2$ operaciones. ¿Cuál de los dos algoritmos es mejor asintóticamente? Determina el valor n_0 que hace a uno mejor que otro para problemas de tamaño $n \geq n_0$.
Note: Usa alguna herramienta que conozcas como Excel para dibujar las funciones.
Para utilizar escala logarítmica en Excel hay que realizar las siguientes operaciones:

- Dibujar el grafico utilizando el tipo de grafico XY(dispersión)
- Una vez terminado el grafico, hacer click sobre el eje x.
- Elegir **Formato de eje** -> **Escala** -> **Escala logarítmica**
- Realizar las mismas operaciones con respecto del eje y

Solución:





1.3

¿Cuáles de las siguientes afirmaciones son ciertas?. Razona tu respuesta.

a) $n^2 \log n \in \Omega(n)$

b) $3n^3 - 10 = \Theta(n^3)$

c) nlogn+4logn es O(nlogn)

Solución:

a. Correcto

$n^2 \log n \geq c.n$

$n^2 \log n \geq n \log n \geq cn$ para $c=1$ y para todo $n_0 \geq 1$.

b. Correcto

$a.n^3 \leq n^3 - 10 \leq b.n^3$

$3n^3 - 10 <= 3n^3$ para $b=3$ y para todo $n_0 \geq 3$.
$3n^3 - 10 >= n^3$ para $a=1$ y para todo $n_0 \geq 3$.

c. Correcto

$n \log n + 4 \log n \leq c.n \log n$

$n \log n + 4 \log n <= n \log n + 4n \log n <= 5n \log n$
para $c=5$ y para todo $n >= 1$

1.4

Demuestra si las siguientes afirmaciones son ciertas o falsas.

A. $5n^2 + 3n \in O(n^2)$
B. $5n^2 + 3n \in \Omega(n^2)$
C. $5n^2 + 3n \in \Theta(n^2)$

Solución:

A. $5n^2 + 3n \leq c.n^2$

$5n^2 + 3n \leq 5n^2 + 3\ n^2 \leq 8.n^2$

Cierto para c=8 y n>=1

B. $5n^2 + 3n \geq c.n^2$

$5n^2 + 3n \geq 5n^2$

Cierto para c=5 y n>=1

C. Es Verdadero porque (A.) es Verdadero y (B.) es Verdadero.

1.5

Dado el siguiente algoritmo que suma los elementos de una matriz cuadrada de orden n, razona cuál es su complejidad.

```
int sum = 0;
for  (int i=0; i< A.length; i++) {
     for(int j=0; j< A.length; i++) {
        sum = sum + A[i,j];
     }
}
```

Solución:

```
int sum = 0;                                    1
for (int i=0; i<n; i++) {                       1;n+1;n
     for (int j=0; j<n; i++) {                  1*n;(n+1)*n;n*n
     sum = sum + A[i,j];                        n*n
}
}
```

Tiempo de la ejecución = $1+1+n+1+n+n+n^2+n+n^2+n^2$

$3n^2+3n+3 \in O(n^2)$

$3n^2+3n+3 \leq c.n^2$ (Verdadero cuando c=9 y $n_0 \geq 1$)

1.6

El número de operaciones ejecutada por el algoritmo A son $8n^2+n$ y por B son $4n^3$. Encuentra n_0 tal que A es mejor que B para todo $n>=n_0$.

Solución:

$8n^2+n<=4n^3$ sii => $n_0>=3$

1.7

Demuestra si las siguientes afirmaciones son ciertas o falsas.

a. $n\log(n) \in \Theta(n)$

b. $2n^2 - 12 \in \Theta(n^2)$

Solución:

a.

$c_1.n \leq n\log(n) \leq c_2.n$

es FALSO para $c_1 = 1$ y $c_2 = 2$ y para todo $n_0 \geq 5$

b.

$c_1.n^2 \leq 2n^2 - 12 \leq c_2.n^2$

es VERDADERO para $c_1 = 1$ y $c_2 = 2$ y para todo $n_0 \geq 4$

1.8

Demuestra que si d(n) es O(f(n)) y e(n) es O(g(n)), no necesariamente se cumple d(n)-e(n) es O(f(n)-g(n)).

Solución:

Contraejemplo:

$d(n) = 4n^2 + 1$, $e(n) = 3n^2$

$f(n)=g(n) = n^2$ => $O(f(n)-g(n))=O(0)$

$d(n)-e(n)= 4n^2 + 1 -3n^2= n^2 + 1$ que es $O(n^2)$, pero no $O(0)$.

1.9

Demuestra que si d(n) es O(f(n)) y e(n) es O(g(n)), entonces d(n)e(n) es O(f(n)g(n))

Solución:

d(n) es O(f(n)) --> existe $n_1>0$ y $c_1>0$ de tal manera que cuando $n>=n_1$, $d(n)<c_1 f(n)$

e(n) es O(g(n) --> existe $n_2>0$ and $c_2>0$ de tal manera que cuando $n>=n_2$, $e(n)<c_2 g(n)$

sea $c=c_1 c_2$ y $n_0=\max\{n_1, n_2\}$, entonces cuando $n>n_0$,

$d(n)e(n)< c_1 f(n) c_2 g(n) = c(f(n)g(n))$ --> d(n)e(n) es O(f(n)g(n))

Capitulo 2

Listas, Pilas y Colas

2.1

Escribe una clase en Java que implementa la siguiente operación:

a) Crear una lista simple enlazada con tamaña 0, y se va incrementando cada vez que se añade un nodo.
b) Obtener el máximo y el mínimo número contenido en una lista simple enlazada

NOTA: Debes implementar tú mismo la lista, es decir, no pueden utilizarse las estructuras de datos propias de Java como ArrayList o Vector.

Solución:

```
public void insertarAlPrincipio(ListNode n) {
	n.next=head;
	head=n;
	size++;
}


/**E-4-B
 * Recorre la lista desde el principio hasta el final para buscar el mínimo
 * Comlejidad O(n)
 * @return
 */
public int buscarMinimo() {
	if (esVacia()) {
		System.out.println("La lista está vacía");
		return -1;
	}

	//Usamos el nodo aux para recorrer la lista
	ListNode aux=head;
```

```
		//Inicializamos el mínimo y el máximo con el valor del primer elemento de la lista
		int min=aux.item;

		while (aux.next!=null) {
			if (min>aux.item) min=aux.item;
			aux=aux.next;
		}


		System.out.println("El mínimo de la lista es " + min);
		return min;
}

/**
 * E-4
 * Recorre la lista desde el principio hasta el final para buscar el máximo
 * Comlejidad O(n)
 * @return
 */
public int buscarMaximo() {
		if (esVacia()) {
			System.out.println("La lista está vacía");
			return -1;
		}

		//Usamos el nodo aux para recorrer la lista
		ListNode aux=head;
		//Inicializamos el mínimo y el máximo con el valor del primer elemento de la lista
		int max=aux.item;

		//Recorremos la lista mientras que no esté vacía
		while (aux.next!=null) {
			if (max<aux.item) max=aux.item;
			aux=aux.next;
		}

		System.out.println("El máximo de la lista es " + max);
		return max;
}
```

2.2

Escribe una clase en Java que ordena una lista simple de enteros de mayor a menor.

NOTA: Utiliza la lista implementada en el problema E-4 para solucionar el problema. Queda prohibido el uso de estructuras de datos propias de Java (ArrayList, Vectors, LinkedList, etc)

Solución:

```
/**
* Dado x, el método crea un nuevo nodo y lo inserta en la posición que le corresponda,
 * teniendo en cuenta que en la lista los elementos se guardan de mayor a menor
 *
```

```
 * */
public void insertarMayorMenor(int x) {
	ListNode newNodo=new ListNode(x);

	if (esVacia()) {
		insertarAlPrincipio(newNodo);
	} else {
		ListNode aux=head;
		ListNode ant=null;
		//Recorremos la lista hasta encontrar la posición
		while (aux!=null && aux.item>x) {
			ant=aux;
			aux=aux.next;
		}

		if (aux==null) {
			//Hemos llegado al final de la lista
			ant.next=newNodo;
		} else if (aux.item<=x) {
			//Debemos insertar antes de aux
			newNodo.next=aux;
			if (ant==null) {
				head=newNodo;
			} else {
				ant.next=newNodo;
			}

		}
		size++;
	}
```

2.3

Escribe una clase en Java para realizar las siguientes operaciones:

a) Buscar un valor en una lista simple.
b) Borrar el valor encontrado o devolver “no encontrado” en el caso de no encontrar el valor.

NOTA: Utiliza la lista implementada en el problema E-4 para solucionar el problema. Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
/**
* Borra el nodo cuyo campo item es igual a v.
 * Si no existe ningún nodo devuelve null
 * @param v
 * @return
 */
public void borrarNodo(int v) {
	//Usamos el nodo aux para recorrer la lista

	if (esVacia()) {
		System.out.println("La lista está vacía. No se puede borrar ningún
nodo");
		return;
```

```
		}
		ListNode aux=head;
		ListNode ant=null;

		while (aux!=null && aux.item!=v) {
			ant=aux;
			aux=aux.next;
		}


		if (aux==null) {
			System.out.println("No Encontrado!!!. La lista no contiene ningún nodo
con valor: " + v);
		} else { //(aux.item==v
			System.out.println("Encontrado!!!. Procedemos a borrar: " + v);
			if (ant==null) {
				head=aux.next; //Tenemos que bor
			} else {
				ant.next=aux.next;
			}
			size--;

		}

}
```

2.4

Crear una clase de Java que implemente una lista doblemente enlazada para almacenar los productos de una fabrica (id, nombre, precio). **La lista debe estar ordenada según id**. Se deberán realizar las siguientes tareas:

- Implementar en Java la clase ***ListaDoblementeEnlazada***
- Implementar el método ***inserta*** un producto en su lugar correspondiente
- Implementar el método ***visualizar*** el listado de todos los productos.

NOTA: Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

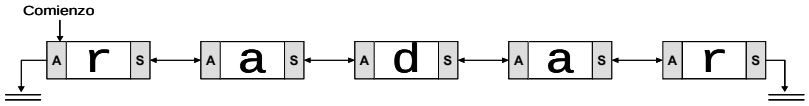
```
/**
 * Como la lista estÃ¡ ordenada en base al id de los productos
 * Deberemos buscar la posiciÃ³n de la lista donde insertar el producto
 * Complejidad O(n), en el peor de los casos tendremos que recorrer la lista
 * @param newNodo
 */
public  void insertarProducto(Producto producto) {
		if (producto==null) {
			System.out.println("El producto es nulo, no podemos insertarlo!!!");
			return; //salimos del mÃ©todo
		}

		//Creamos un nodo para almacenar ese producto
		DNode<Producto> newNodo=new DNode<Producto>(producto,null,null);
			if (size==0) {
```

```
                //La lista estÃ¡ vacÃ-a, por tanto podemos usar el mÃ©todo insertarInicio
                insertarInicio(newNodo);
        } else {
                //Miramos a ver si tenemos que insertar al principio
                if (header.next.item.id>producto.id) insertarInicio(newNodo);
                else {
                        //Miramos a ver si tenemos que insertar al final, justo antes de
tailer
                        if (tailer.prev.item.id<producto.id) insertarAntes(tailer,newNodo);
                        else {
                                DNode<Producto> auxiliar=header.next;
                                //Recorremos la lista hasta el final o bien hasta encontrar
la posiciÃ³n
                                //donde debemos insertar
                                while (auxiliar!=tailer && auxiliar.item.id<producto.id) {
                                        auxiliar=auxiliar.next;
                                }

                                /**Tenemos que insertar justo antes del nodo Auxiliar*/
                                insertarAntes(auxiliar,newNodo);
                        }

                }

        }

}

/**
 * MÃ©todo que recorre lalista de productos para mostrarlos.
*/
public void showProductos() {
        DNode<Producto> auxiliar=header.next;
        //Recorremos la lista hasta el final o bien hasta encontrar la posiciÃ³n
        //donde debemos insertar
        while (auxiliar!=tailer) {
                auxiliar.item.view();
                auxiliar=auxiliar.next;
        }
        System.out.println();
}

package sheet1.EJ07y08;


public class Producto {
int id;
String nombre;
float precio;

/**
 * Método constructor
 * @param i
 * @param n
 * @param p
 */
public Producto(int i, String n, float p) {
        id=i;
```

```
		nombre=n;
		precio=p;
		}

/**
 * Método que muestra los datos de un producto
 */
public void view() {
		System.out.println(id+"-"+nombre+":"+precio);
		}
}
```

2.5

Un palíndromo es una palabra que se lee igual en ambos sentidos, como por ejemplo radar. Diseña un método en Java que determine si una palabra definida como una lista doblemente enlazada de caracteres es palíndroma o no.



NOTA: Utiliza la lista implementada en el problema E-7 para solucionar el problema. Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
/**
* Comprueba si una palabra es palindroma o no
* @param lWord: recibe una lista doble enlazada de carácteres
* @return
*/
public  boolean esPalindroma() {
/** NOdo que nos ayuda a recorrer la lsita de izquierda a derecha*/
DNode<Character> fromLeft=header.next;
/** NOdo que nos ayuda a recorrer la lsita de derecha a izquierda*/
DNode<Character> fromRight=tailer.prev;
/**
* Paramos cuando el nodo izquierdo ha sobrepasado al derecho o cuando los valores son distintos
*/
while (fromRight.next!=fromLeft.prev && fromLeft.prev!=fromRight  &&
fromLeft.item==fromRight.item) {
fromLeft=fromLeft.next;
fromRight=fromRight.prev;
}
/**
* Si se cumple está condición quiere decir que el nodo izquierdo no ha sobrepasado al derecho y por tanto,
* existe algún valor que es distinto.
*/
if (fromRight.next!=fromLeft.prev && fromLeft.prev!=fromRight) return false;
else return true;
}
```

2.6

a) ¿Qué diferencia hay entre una lista simple una lista circular?

b) Implemente en Java:

- Una clase que implemente una lista circular.
- Un método que cuente el número de nodos existentes en una lista circular.

NOTA: Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

La principal diferencia entre una lista circular y una lista simple, es que el último nodo de una lista circular apunta al primer nodo de la lista, mientras que en una lista simple, el último nodo apunta a null.

```
public class Node<E> {
	public E item;
	public Node<E> next;

	public Node(){
		item=null;
		next=null;
	}

	public Node(E objecto, Node<E> n){
		item=objecto;
		next=n;
	}
	public Node(E objecto){
		item=objecto;
		next=null;
	}
}

public class SListCircular<E> {
	private Node<E> header;
	private int size;

	/** Crea una lista vacia*/
	public SListCircular() {
		header=new Node<E>();
		header.next=null;
		size=0;
	}

	public boolean esVacia() {
		return (size==0);
		//return (header.next==null)
	}



	/**
	 * Devuelve el número nodos de la lista
```

```
 * @return
 */
public int getSize() {
	return size;
}


/**
 * Remove the first node
 * Complexity O(1).
 */
public void removeFirst() {
	if (esVacia()) {
		System.out.println("The list is empty!!!");
		return;
	}

	Node<E> oldFirst=header.next;
	//Now, header must be the following node in the list
	header.next=oldFirst.next;
	//decrease size
	size--;
}

/**
 * Inserts the newNodo at the beginning of the list
 * Complexity O(1)
 * @param newNodo
 */
public  void insertarFirst(Node<E> newNodo) {
	if (newNodo==null) {
		System.out.println("The node is null, we cannot insert it!!!");
		return;
	}

	Node<E> first=header.next;
	//Now, the first node must be the second one.
	newNodo.next=first;
	header.next=newNodo;

	//If the list is empty, the new first node should reference to header.
	if (first==null) newNodo.next=header;

	size++;

}


/**
 * Remove the last node in the list
 * We must trasverse all nodes until to find the penultimum node (its next is null).
 * Complexity O(n), porque el número de operaciones depende del
 * tamaño de la lista (ya que debemos visitar sus n nodos).
 */
public void removeLast() {

	if (esVacia()) {
		System.out.println("The list is empty!!!");
```

```
                return; //salimos del método
            }

            Node<E> aux=header.next;

            Node<E> penultimum=header;
            //Trasverse the list to find the last node
            while (aux.next!=header) {
                penultimum=aux;
                aux=aux.next;
            }


            if (penultimum==header) {
                header.next=null;//now the list is empty
            } else {
                //The list has two or more elements
                penultimum.next=header;
            }
            size--;


        }

        /**
         *
         * Inserts the newNodo at the end of the list
         * Complexity O(1), since we only run a constant number of instructions
         * @param newNodo
         */
        public void insertarLast(Node<E> newNodo) {
            if (newNodo==null) {
                System.out.println("The node is null, we cannot insert it!!!");
                return;
            }
            if (esVacia()) insertarFirst(newNodo);
            else {
                Node<E> auxiliar=header.next;
                //El nodo que apunte a header es el último, y por tanto debemos
parar
                while (auxiliar.next!=header) {
                    auxiliar=auxiliar.next;
                }
                //hemos llegado al último nodo, ahora su referencia next
                //debe ser newNodo
                auxiliar.next=newNodo;
                //newNodo.next debe ser header, porque es el final de la lista
                newNodo.next=header;
                //incrementamos el tamaño
                size++;
            }
        }
```

```
	/**Trasverses the nodes in the list and shows them*/
	public void show() {
		if (esVacia()) {
			System.out.println("The list is empty");
			return;
		}

		Node<E> auxiliar=header.next;
		System.out.print(auxiliar.item+",");
		//We traverse the nodes until to return to the first node
		while (auxiliar.next!=header) {
			auxiliar=auxiliar.next;
			System.out.print(auxiliar.item+",");
		}
		System.out.println();
	}
	/**
	 * Method to test the class
	 * @param args
	 */
	public static void main(String args[]) {
		SListCircular<Integer> lista=new SListCircular<Integer>();
		for (int i=0; i<5; i++) {
			Node<Integer> newNodo=new Node<Integer>(i);
			lista.insertarFirst(newNodo);
		}
		lista.show();
		for (int i=5; i<10; i++) {
			Node<Integer> newNodo=new Node<Integer>(i);
			lista.insertarLast(newNodo);
		}

		lista.show();

	}
}
```

2.7

Dadas las operaciones presentes en la tabla (ejecutadas en el orden en el que aparecen), ¿cuáles son las salidas y el contenido de la pila después de realizar cada operación? Justifica tu respuesta.

Solución:

Operación	*Salida*	*Contenido*	*Justificación*
push(4)	-	4	Apilar nodo
push(1)	-	1, 4	Apilar nodo
pop()	1	4	Desapilar nodo
push(6)	-	6, 4	Apilar nodo
pop()	6	4	Desapilar nodo
top()	4	4	Desapilar nodo
pop()	4	-	Ultimo nodo

pop()	error	-	Pila vacía
isEmpty()	true	-	Pila vacía
Push(8)	-	8	Apilar nodo
Push(6)	-	6, 8	Apilar nodo
Push(1)	-	1, 6, 8	Apilar nodo
Push(4)	-	4, 1, 6, 8	Apilar nodo
Size()	4	4, 1, 6, 8	Tamaño
Pop()	4	1, 6, 8	Desapilar nodo
Push(9)	-	9, 1, 6, 8	Apilar nodo
Pop()	9	1, 6, 8	Desapilar nodo
Pop()	1	6, 8	Desapilar nodo

2.8

Dada la función recursiva factorial(), desarrollar una clase Java que simule el crecimiento / dcrecimiento de la pila a medida que se hacen las llamadas recursivas.

Solución:

```
public static int factorialRec(int n) {
		stackCalls.push("factorial("+(n)+")");
		if (n<=0) {
			//stackCalls.pop();
			return 1;
		}
		else {
			int res=n*factorialRec(n-1);
			stackCalls.pop();

			//stackCalls.push("factorial("+(n-1)+")");
			for (int i=stackCalls.size()-1;i>=0;i--)
System.out.println(stackCalls.get(i));
			System.out.println();
			return res;
		}
	}
```

2.9

Implementa en Java un programa que utiliza una pila para comprobar los paréntesis de una expresión aritmética como [(5+x)-(y+z)], según los pasos siguientes:

1. Si encontramos un símbolo de apertura (**[**,**(**,**{**), debemos lo apilamos.
2. Si encontramos un símbolo de cierre (**]**,**}**,**)**), entonces consultamos el elemento que hay en la cima de pila. Si son del mismo tipo, sólo entonces debemos desapilamos.

La expresión estará balanceada si al terminar de leer la expresión la pila está vacía.

NOTA: Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
public boolean checkParenthesis(char[] c) {
		LinkedListStack<Character> chars = new LinkedListStack<Character>();
		for(int i=0;i<c.length;i++){
			if (c[i]=='[' || c[i]=='(' || c[i]=='{'){
					chars.push(new Character(c[i]));
			}else if (c[i]==']' || c[i]==')' || c[i]=='}'){
				char aux = chars.top().charValue();
				if ((c[i]==']' && aux=='[')||
					(c[i]==')' && aux=='(')||
					(c[i]=='}' && aux=='{')){
					chars.pop();
				}
			}
		}
		//Ahora que tenemos la lista hacemos la comprobaci�n.
		if (chars.isEmpty()){
			return true;
		}else{
			return false;
		}
	}
```

2.10

Escribir un programa en Java que reciba una expresión matemática en formato postfijo y con la ayuda de una pila obtener el resultado de evaluar dicha expresión. La expresión matemática en formato postfijo la recibirá el algoritmo a través de un array, suponiendo que dicha expresión siempre estará bien formulada y los operandos serán valores comprendidos entre 0 y 9.
Ejemplo:
Notación postfija: **45*46+/**

Resultado del programa: **2**

NOTA: Utiliza la lista implementada en el problema E-12 para solucionar el problema. Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
public int postFixedFormat(char[] c) {
		LinkedListStack<Integer> ints = new LinkedListStack<Integer>();
		for(int i=0;i<c.length;i++){
			if (c[i]=='*'){
			   int v1 = ints.pop().intValue();
			   int v2 = ints.pop().intValue();
				ints.push(new Integer(v2*v1));
			} else if (c[i]=='+') {
			   int v1 = ints.pop().intValue();
			   int v2 = ints.pop().intValue();
				ints.push(new Integer(v2+v1));
```

```
                } else if (c[i]=='-') {
                    int v1 = ints.pop().intValue();
                    int v2 = ints.pop().intValue();
                        ints.push(new Integer(v2-v1));
                } else if (c[i]=='/') {
                    int v1 = ints.pop().intValue();
                    int v2 = ints.pop().intValue();
                        ints.push(new Integer(v2/v1));
                } else {
                        ints.push(new Integer(Character.digit(c[i],10)));
                }
            }
            return ints.pop();
        }
```

2.11

La Sociedad Cooperativa TEIDE-HEASE necesita para su gestión de personal una estructura de datos que permita almacenar los datos de todos sus empleados. De cada empleado será necesario conocer su nombre, apellidos, DNI, dirección y teléfono. A dicha estructura de datos será necesario que se vayan añadiendo empleados según se les va contratando, y se irán eliminando cuando se les despida. El Departamento de Recursos Humanos del centro tiene la política de despedir siempre al empleado que menos tiempo lleve en el centro.

Se pide:
Implementar una clase en Java que use la estructura y las operaciones anteriores para realizar los siguientes algoritmos:

- El algoritmo **contratar(p)** que permita contratar a un empleado. Indicar la complejidad que tendría este proceso. (**p** es un nodo de tipo empleado).
- El algoritmo **despedir()** que permita despedir a un empleado. Indicar la complejidad que tendría este proceso.
- Si el Departamento de Recursos Humanos del centro deseara modificar su política de despido y despedir a un trabajador cualquiera (por su DNI), ¿cómo sería la complejidad de dicho proceso? Implementa el método **despedir(dni)** propuesto.

Solución:

```
public void hire(Worker e){
            workers.push(e);
        }

        public Worker fire(){
            if(workers.isEmpty()){
                System.out.println("There is no worker at the enterprise");
                return null;
            }
            return workers.pop();
        }

        public Worker fireByNIC(String nic){
```

```
		if(workers.isEmpty()){
			System.out.println("There is no worker at the enterprise");
			return null;
		}

		Worker aux = null;
		LinkedListStack<Worker> pilaAux = new LinkedListStack<Worker>();
		boolean found = false;

		while(!workers.isEmpty() || !found){
			aux = workers.pop();
			if(aux.nic.equals(nic)){
				found = true;
			}
			else {
				pilaAux.push(aux);
			}
		}

		if(!found){
			System.out.println("There is no worker with this NIC");
			return null;
		}

		while(!pilaAux.isEmpty()){
			workers.push(pilaAux.pop());
		}

		return aux;
	}


public class Worker {

	String name;
	String surname;
	String nic;
	String address;
	String phonenumber;

	public Worker(String name, String surname, String nic, String address,
			String phonenumber) {
		super();
		this.name = name;
		this.surname = surname;
		this.nic = nic;
		this.address = address;
		this.phonenumber = phonenumber;
	}

}
```

2.12

Dadas las siguientes operaciones realizadas sobre una cola vacía, escriba las salidas y el contenido de la cola después de realizar cada operación y justifica tu respuesta.

Solución:

Operación	*Salida*	*Contenido*	*justificación*
enqueue(4)	-	4	encolar nodo
enqueue(2)	-	2, 4	encolar nodo
dequeue()	4	2	Desencolar nodo
enqueue(8)	-	8, 2	encolar nodo
dequeue()	2	8	Desencolar nodo
front()	8	8	Ver la frente
dequeue()	8	-	Desencolar nodo
dequeue()	error	-	Desencolar nodo
isEmpty()	true	-	Cola esta vacía
enqueue(6)	-	6	encolar nodo
enqueue(8)	-	8, 6	encolar nodo
size()	2	8, 6	tamaño
enqueue(2)	-	2, 8, 6	encolar nodo
enqueue(4)	-	4, 2, 8, 6	encolar nodo
dequeue()	6	4, 2, 8	Desencolar nodo

2.13

Disponemos de dos colas, ‘cola1’ y ‘cola2’, en las que se han insertado elementos de forma ordenada. Por ejemplo, cola1=[2,3,6,8,9], cola2=[0,1,4,5,7].
Se pide:

a) Implementa un método iterativo en Java que mezcla ambas colas en otra cola ‘cola3’ ordenada.

b) Calcula y justifica brevemente la complejidad de la función.

NOTA: Utiliza la lista implementada en el problema E-15 para solucionar el problema. Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
public LinkedListQueue<Integer> mixQueueIterative(LinkedListQueue<Integer> c1,
                LinkedListQueue<Integer> c2){
LinkedListQueue<Integer> aux = new LinkedListQueue<Integer>();
while(!c1.isEmpty() || !c2.isEmpty()){
        if(c1.isEmpty()){
                aux.enqueue(c2.dequeue());
```

```
                }
                else if(c2.isEmpty()){
                        aux.enqueue(c1.dequeue());
                }
                else{
                        if(c1.front().intValue()>c2.front().intValue()){
                                aux.enqueue(c2.dequeue());
                        }
                        else{
                                aux.enqueue(c1.dequeue());
                        }
                }
        }
        return aux;
}
```

2.14

Implemente de nuevo el método del ejercicio anterior utilizando para ello recursividad.

Solución:

```
public LinkedListQueue<Integer> mixQueueRecursive(LinkedListQueue<Integer> c1,
                                LinkedListQueue<Integer> c2){
        LinkedListQueue<Integer> aux = null;
        if(c1.isEmpty() && c2.isEmpty()){
                aux = new LinkedListQueue<Integer>();
                return aux;
        }
        Integer number;
        if(c1.isEmpty()){
                number = c2.dequeue();
                aux = mixQueueRecursive(c1, c2);
                aux.enqueue(number);
        }
        else if(c2.isEmpty()){
                number = c1.dequeue();
                aux = mixQueueRecursive(c1, c2);
                aux.enqueue(number);
        }
        else{
                if(c1.front()>c2.front()){
                        number = c2.dequeue();
                        aux = mixQueueRecursive(c1, c2);
                        aux.enqueue(number);
                }
                else{
                        number = c1.dequeue();
                        aux = mixQueueRecursive(c1, c2);
                        aux.enqueue(number);
                }
        }
        return aux;
}
```

2.15

Implementa la clase Cola en Java y utilízala posteriormente para implementar la siguiente función:
La función consiste en ir eliminando de una cola el elemento que ocupa la posición *k,* pasando al final de la cola los *k-1* que le preceden. El proceso se repite hasta que solo quede un elemento en la cola.

Por ejemplo, con k=3, en la siguiente cola:

Cola = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

en el primer paso pasaríamos el 1 al final, el 2 también y borraríamos el elemento 3 quedando:

Cola = | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 |

en el segundo paso pasaríamos el 4 al final, el 5 también y borraríamos el elemento 6 quedando:

Cola = | 7 | 8 | 9 | 1 | 2 | 4 | 5 |

en el tercer paso pasaríamos el 7 al final, el 8 también y borraríamos el elemento 9 quedando:

Cola = | 1 | 2 | 4 | 5 | 7 | 8 |

en el cuatro paso pasaríamos el 1 al final, el 2 también y borraríamos el elemento 4 quedando:

Cola = | 5 | 7 | 8 | 1 | 2 |

en el quinto paso pasaríamos el 5 al final, el 7 también y borraríamos el elemento 8 quedando:

Cola = | 1 | 2 | 5 | 7 |

en el sexto paso pasaríamos el 1 al final, el 2 también y borraríamos el elemento 5 quedando:

Cola = | 7 | 1 | 2 |

en el séptimo paso pasaríamos el 7 al final, el 1 también y borraríamos el elemento 2 quedando:

Cola = | 7 | 1 |

en el octavo paso pasaríamos el 7 al final, el 1 también y borraríamos el elemento 7 quedando:

Cola = | 1 |

en el noveno paso pasaríamos el 1 al final, y otra vez el 1 al final, luego borraríamos el 1 quedando la cola vacía:

Cola = | |

Devolver el (1)

NOTA: Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
public int josephus(int x) {
		int n=0;
		while (!queue.isEmpty()){
			for (int i = 0; i<x; i++){
				queue.enqueue(queue.dequeue());
			}
			n = queue.dequeue();
			show(queue);
		}
		return n;
	}
```

2.16

Dadas dos pilas S y T (no están vacías), y una cola D de tipo (Double-Ended Queues). Escribe un programa en Java que utiliza D para que S guarde bajo sus elementos todos los elementos de T.

NOTA: Para obtener más información sobre Double-Ended Queues consultar el libro de Goodrich pagina 213.

Solución:

```
public static void mount(LinkedListStack<Integer> s, LinkedListStack<Integer> t) {

		if (s.isEmpty()||t.isEmpty()) return;

	DQueue<Integer> D=new DQueue<Integer>();


	while (!s.isEmpty()) {

		DNode<Integer> obj=new DNode<Integer>(s.pop());

		D.insertLast(obj);
```

```
        }
        System.out.print("D ");
        D.show();

        while (!t.isEmpty()) {

                DNode<Integer> obj=new DNode<Integer>(t.pop());

                D.insertLast(obj);

        }
System.out.print("D ");

D.show();

System.out.println();
while (!D.isEmpty()) {
//D.show();
s.push(D.removeLast());
}
System.out.println("S ");
while (!s.isEmpty()) {
        System.out.println(s.pop());
        }
}
```

2.17

Implementa en Java una clase para mostrar cómo usar una pila y una cola para generar todos los subconjuntos posibles de un conjunto de n-elementos T, sin recursividad.

Por ejemplo, si T={A,B,C}, entonces todos sus posibles subconjuntos son: {}, {A}, {B}, {C}, {A,B}, {A,C}, {B,C}, y {A,B,C}.

NOTA: Queda prohibido el uso de estructuras de datos propias de Java.

Solución:

```
static LinkedListQueue<String> powerset(String set) {

        LinkedListStack<String> s = new LinkedListStack<String>();
        LinkedListQueue<String> q = new LinkedListQueue<String>();

        q.enqueue("");

        for (int i = 0; i < set.length(); i++) {

                while (!q.isEmpty()) {
                        String subset = q.dequeue();
                        s.push(subset);
                        s.push(subset + set.charAt(i));

                }
                while (!s.isEmpty()) {
                        String str = s.pop();
                        q.enqueue(str);
```

```
                }
            }
            return q;
        }
```

2.18

¿Qué estructura de datos recomiendas para implementar una cola y que sus operaciones **queue(e)** y **deque()** tengan complejidad O(1). Razona tu respuesta.

Solución:

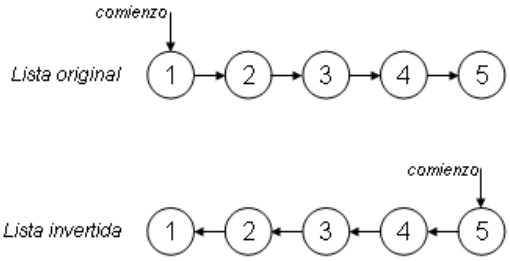
Es suficiente con una lista simple enlazada con dos propiedades head (primer nodo) y tail (último nodo), queue=insertarLast O(1), dequeu=removeFirst O(1)

2.19

Dada el siguiente fragmento de una clase que implementa una lista simple enlazada:

```
public class ListaSimple {
 Nodo comienzo;
 int tamaño;
 public ListaSimple() {
  comienzo=null;
  tamaño =0;
}
public boolean esVacia() { }
/** Devuelve el valor almacenado en el primer nodo de la lista. Si
la lista es vacía, entonces el método lanza una excepción de tipo
ListaVaciaException */
public int consultaPrimero() throws ListaVaciaException{ ….}
/** Devuelve el valor almacenado en el ultimo nodo de la lista. Si
la lista es vacía, entonces el método lanza una excepción de tipo
ListaVaciaException */
public int consultaUltimo() throws ListaVaciaException{ ….}
public void insertarPrincipio(Nodo  n) { ….}
public void insertarFinal(Nodo n) { ….}
public Nodo borrarPrimero( ) { ….}
public Nodo borrarUltimo( ) { ….}
/** Devuelve el primer nodo con valor x; si no lo encuentra
devuelve null */
public Nodo buscaNodo(int x) { ….}
/** Busca y borra el nodo de la lista cuyo valor sea x */
public void borrarNodo(int x) { ….} … }
```

Completa el siguiente método recursivo *invertirLista* de la clase *ListaSimple* que permita invertir la dirección de los enlaces de una lista enlazada simple. El orden de los enlaces debe ser el opuesto al original, como se muestra en el ejemplo:




```
public void invertirLista() {
      if (esVacia ()) return;
      else {
            Nodo primero=
            ....
            invertirLista ();
            …

            …
      }
}
```

Solución:

```
public void invertirLista() {
      if (esVacia()) return;
      else {
            Nodo primero=comienzo;
            borrarPrimero();
            invertirLista ();
            insertarUltimo(primero);
      }
}
```

2.20

Dado el fragmento de código anterior, completa el siguiente método que reciba como parámetros dos listas simples de enteros ordenados de menor a mayor y que devuelva una nueva lista simple enlazada como unión de ambas con sus elementos ordenados de la misma forma.

Solución:

```
public static ListaSimple  mix(ListaSimple  L1, ListaSimple  L2) {
      ListaSimple  L3=new ListaSimple ();
      while (L1.esVacia()==false && L2.esVacia()==false) {
            if (L1.consultaPrimero()<L2.consultaPrimero()) L3.
insertarPrincipio(L1.borrarPrimero());
            else if (L2.consultaPrimero()<L1. consultaPrimero())
L3. insertarPrincipio (L2.borrarPrimero());
            else {
                  L3. insertarPrincipio (L1.borrarPrimero());
                                    L3. insertarPrincipio
(L2.borrarPrimero());
            }
```

```
	}

	while (L1.esVacia()==false) L3. insertarPrincipio
(L1.borrarPrimero());
                  while (L2.esVacia()==false) L3.
insertarPrincipio (L2.borrarPrimero());

	return L3;
}
```

2.21

Completa la siguiente clase que implemente una cola utilizando un array circular. Debes completar los métodos size(), isEmpty(), dequeue(), enqueue(E e) y front().

```
public class ArrayCircularCola<E> {
	/**array que almacena los elementos de la cola*/
	protected E queue[];
	/**índice que almacena la posición en el array del próximo
elemento en salir*/
	protected int front;
	/**índice que indica la posición en el array donde se debe
encolar  el   próximo elemento*/
	protected int tail;

	/**  Creamos un cola vacía, reservamos memoria para un array
de maxElem
	 * @param maxElem
	 */
	public ArrayQueue(int maxElem){
		queue=(E[]) new Object[maxElem];
		front=0;
		tail=0;
	}
		/**Devuelve el tamaño de la cola*/
		    public int size() {

		    }

		public boolean isEmpty() {

		      }

	/**Operación desencolar*/
	public E dequeue() {

	}

	public void enqueue(E e) {

	}

	public E front() {

	}
}
```

Solución:

```
public class ArrayCircularCola<E> {
	/**array que almacena los elementos de la cola*/
	protected E queue[];
	/**índice que almacena la posición en el array del próximo elemento en salir*/
	protected int front;
	/**índice que indica la posición en el array donde se debe insertar (encolar) el   próximo elemento*/
	protected int tail;

	/**  Creamos un cola vacía, reservamos memoria para un array de maxElem
	 * @param maxElem
	 */
	public ArrayQueue(int maxElem){
		queue=(E[]) new Object[maxElem];
		front=0;
		tail=0;
	}
		/**Devuelve el tamaño de la cola*/
		 public int size() {
		return (queue.length - front + tail) % queue.length;
	}

	/**Operación desencolar*/
	public E dequeue() {
		if (isEmpty()) {
				System.out.println("Queue is empty");
					return null;
		}
		E temp=queue [front];
		front = (front + 1) % queue.length;
		return temp;
	}

	public void enqueue(E e) {
		if (size() == queue.length-1) {
			System.out.println("Queue is full");
			return;
		}
		queue [tail] = e;
		tail = (tail + 1) % queue.length;

	}
	public E front() {
		if (isEmpty()) {
				System.out.println("Queue is empty");
					return null;
		}
		return queue [front];
	}

	public boolean isEmpty() {    return (front==tail);   }

	}
```

2.22

a) Razona detalladamente si la siguiente afirmación es verdadera o falsa.
Se utiliza una centinela Tailer al final de cualquier lista (tanto simple como doble) enlazada para que la inserción al final siempre tiene complejidad O(1).

b) ¿Qué estructura de datos recomiendas para seguir una estrategia de **First In-Last Out (FILO)** sobre sus nodos. Por favor, indica y razona la complejidad de cada una de sus operaciones principales.

c) ¿Qué diferencia existe entre una cola y una lista simple?, ¿y entre una cola normal y una Double-ended Queue?.

Solución:

a) En el caso de la SList con variable tail, las inserciones al final tienen conste O(1), ya que la operación se puede realizar en un numero constantes de instrucciones (tail.next=newNodo; newNodo.next=null;tail=newNodo).

b) Es una pila y la complejidad es push() O(1), pop() O(1)

c) Una cola sigue la estrategia FIFO y tiene dos operaciones encolar () O(1) y desencolar() O(1), mientras la lista normal no sigue ningún estrategia y se puede insertar y borrar del principio, del medio y del final. Las dobles colas permiten inserciones y borrados tanto al principio como al final, mientras que las colas simples solo permiten insertar al final y borrar del principio.

2.23

Dada el siguiente fragmento de la clase que implementa una lista simple enlazada:

```
public class SList<E> {
	/**Es el primer nodo de la lista*/
	Node<E> head;
	int size;
	public SList() {
		head=null;
		size=0;
	}
		public boolean esVacia() { }
		public void buscarNodo(Node<E> n) { ….}
}
```

¿Es correcto el siguiente código?. En caso de no ser correcto, corrige y completa el código para que lo sea.

```
public booblen buscarNodo(Node<E> n) {
	if (esVacia()) {
		System.out.println("La lista está vacía");
		return -1;
	}
```

```
	//Recorremos la lista mientras que no esté vacía
	while (aux != null && aux.item != n.item) {
		aux = n.next;
	}
	return aux;
}
```

Solución:

En primer lugar debemos inicializar la variable aux: Node<E> aux=head; Además, también hay que corregir (aux = n.next) a (aux=aux.next), para que no entre en un bucle infinito.

2.24

Dada el siguiente fragmento de la clase que implementa una lista doble enlazada:

```
public class DList<E> {
	/**Nodo centinela cuyo propiedad next apunta al primer nodo
de la lista*/
	public DNode<E> header;
	/**Nodo centinela cuyo propiedad prev apunta al ultimo nodo
de la lista*/
	public DNode<E> tailer;
	public  int size;
	/** Crea una lista vacía. Para ello inicializa los dos nodos
centinelas. */
	public DList() {
		header=new DNode<E>();
		tailer=new DNode<E>();
		//Se apuntan el uno al otro
		header.next=tailer;
		tailer.prev=header;
		size=0;
	}
	public boolean isEmpty() { }
		public void insertarFirst(DNode<E> n) { ….}
		public void insertarLast(DNode<E> n) { ….}
		public void removeFirst(DNode<E> n) { ….}
		public void removeLast(DNode<E> n) { ….}
				….
}
```

Escribe el código del método que inserta el último elemento de la lista y razona su complejidad.

Solución:

```
public  void insertLast(DNode<E> newNodo) {
		if (newNodo==null) {
			System.out.println("El nodo es nulo, no podemos
insertarlo!!!");
			return; //salimos del método
		}
```

```
            DNode<E> oldLast=tailer.prev;
            oldLast.next=newNodo;
            newNodo.prev=oldLast;
            newNodo.next=tailer;
            tailer.prev=newNodo;

            //Incrementamos el tamaño de la lista
            size++;

      }
```

Complejidad O(1), porque solo ejecuta un numero constante de instrucciones.

2.25

A partir de la implementación propuesta en el ejercicio 2.24 para una lista doble enlazada, escribe los métodos **insertarAntes**(DNode<E> v, DNode<E> newNodo) que inserte el nodo newNodo justo antes del nodo v y el método **borrar**(DNode<E> v) que borre el nodo de la lista.

Solución:

```
public void insertarAntes(DNode<Producto> v, DNode<Producto>
newNodo) {
            if (v==header) {
                  System.out.println("No puedo insertar antes del
nodo centinela.");
                  return;
            }

            DNode<Producto> aux=v.prev;

            newNodo.prev=aux;
            aux.next=newNodo;

            newNodo.next=v;
            v.prev=newNodo;

      }

public void remove(DNode<E> v) {
            if (v==header || v==tailer) {
                  System.out.println("Not to remove a sentinel
node.");
                  return;
            }
            DNode<E> ant=v.prev;
            DNode<E> sig=v.next;
            sig.prev=ant;
            ant.next=sig;
            size--;
      }
```

2.26

A partir de la implementación propuesta en el ejercicio 2.24, desarrolla un método recursivo (de la clase DList) que invierta sus nodos. Por ejemplo, si la lista es L={0,1,2}, al invertir los nodos de la lista, el resultado será L={2,1,0}. Ojo, se invierten los nodos de la lista, no sus valores.

Solución:

```
public void reverseRec() {
            if (isEmpty ()) return;
            else {
                  DNode<E> firstNode=header.next;
                  removeFirst();
                  reverseRec();
                  insertLast(firstNode);
            }
      }
```

2.27

Dada la siguiente implementación de una pila utilizando una lista enlazada **que no utiliza nodos centinelas**:

```
import DNode;
public class LinkedListStack<E>  {
      protected Node<E> top;
      /**Stores the number of elements of the stack*/
      protected int size=-1;

      /**Constructor. It creates an empty stack*/
      public LinkedListStack() {…}
      public E pop(){...}
      public E top(){...}
      public void push(E e) {...}
      public int size() { return (size); }
      public boolean isEmpty() { return (size<=0);   }
      /**This methods prints the elements of the stack*/
      public String toString() {...}
}
```

Escribe un método buscarenPila que reciba una pila de enteros y devuelva un boolean que determine si la pila contiene el valor v o no. Se recomienda utilizar una pila auxiliar.

Solución:

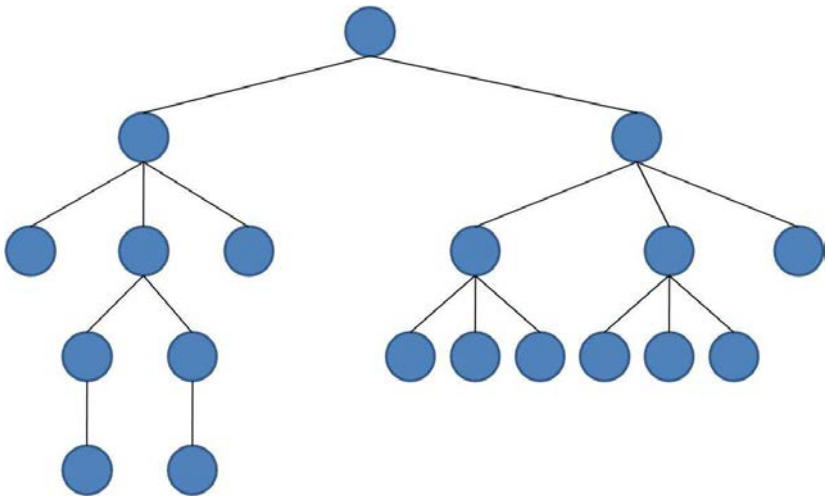
```
static boolean buscarenPila (LinkedListStack<Integer> Pila, int
v){
            if(Pila.isEmpty()){
                  System.out.println("Pila is Empty ");
                  return false;
            }
```

```
            Node<Integer>  aux = null;
            LinkedListStack<Integer> pilaAux = new
LinkedListStack< Integer >();
            boolean found = false;

            while(!Pila.isEmpty() || !found){
                  aux = Pila.pop();
                  if(aux.item = v){
                        found = true;
                  }
                  else {
                        pilaAux.push(aux);
                  }
            }

            while(!pilaAux.isEmpty()){
                  Pila.push(pilaAux.pop());
            }

            if(!found){
                  System.out.println("Value is not found");
                  return false;
            } return true;



      }
```

Capitulo 3

Árboles

3.1



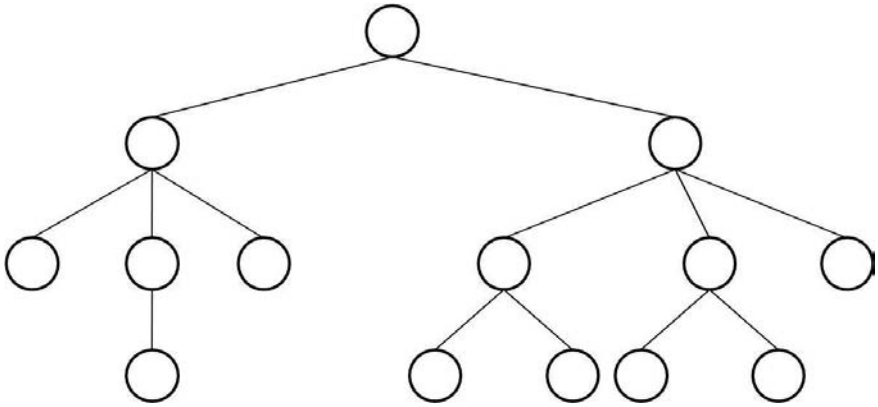
a) Explica los valores de las principales características del árbol mostrado en la figura. NOTA: Estas características son grado del árbol, altura del árbol, número nodos del árbol, hojas y nodos internos.
b) Dadas las siguientes propiedades de un árbol, proporcione un dibujo que satisfaga las mismas:
    1. Grado del árbol: 3
    2. Nº de Nodos: 14
    3. Altura del árbol: 3
    4. Nº nodos con profundidad=2: 6

Solución:

a) Grado del árbol: 3
   Altura del árbol: 4

Nodos del árbol: 19
Nodos Internos: 8
Nodos externos: 11

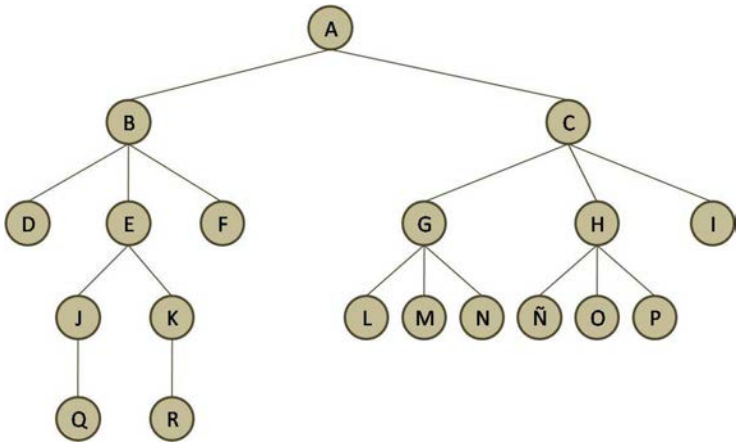
b)



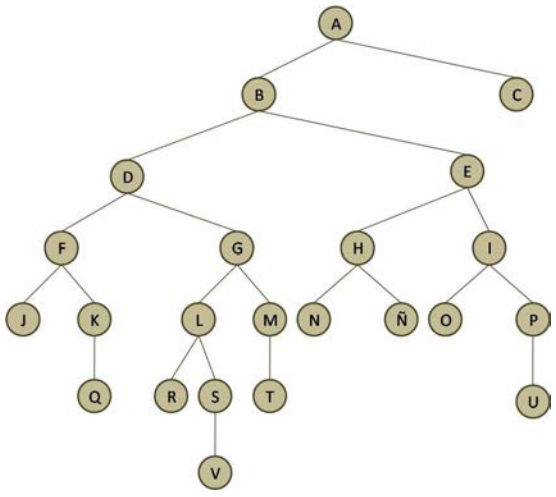
3.2

Dados los siguientes árboles, escriba los recorridos pre-orden, in-orden y post-orden de cada uno de ellos.

a)



b)



Solución:

a) Pre-orden: ABDEJQKRFCGLMNHÑOPI
In-orden: No tiene porque no es binario.
Post-orden: DQJRKEFBLMNGÑOPHICA

b) Pre-orden: ABDFJKQGLRSVMTEHNÑIOPUC
In-orden: JFQKDRLVSGTMBNHÑEOIUPAC
Post-orden: JQKFRVSLTMGDNÑHOUPIEBCA

3.3

Implemente un método en java que devuelva el máximo valor almacenado en un árbol binario de enteros. Si el árbol está vacío, devolverá -1.

Solución:

```
public int maximunValue(){
	if(this.isEmpty()){
		System.out.println("Tree empty");
		return -1;
	}
	int max = 0;
	BinaryNode<E> aux = this.root;
	while(aux!=null){
		if(aux.value instanceof Integer){
			if((Integer)aux.value>max){
				max = ((Integer)aux.value).intValue();
			}
		}
		else{
			System.out.println("Only applicable to Integer
values Tree");
			return -1;
		}
		aux = this.preorderNext(aux);
	}
	return max;
}
```

3.4

Implementa un algoritmo en java para calcular el número de descendientes de cada nodo de un árbol binario. El algoritmo debe ser recursivo.

Solución:

```
public int calculateDescendents(BinaryNode<E> n){
	int right = 0;
	int left = 0;
	if(n.leftChild!=null){
		left = calculateDescendents(n.leftChild)+1;
	}
```

```
        if(n.rightChild!=null){
                right = calculateDescendents(n.rightChild)+1;
        }
        return (left+right);
}
```

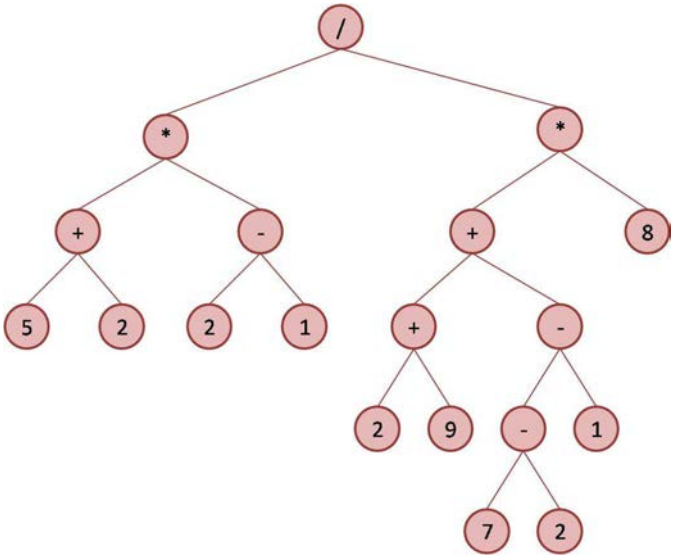
3.5

Dibuje la representación de un árbol binario de la siguiente expresión aritmética:

"(((5 + 2) * (2 – 1))/((2 + 9) + ((7 – 2) – 1) * 8)"

Implementar en java el método que realiza dicha representación de la expresión a partir del árbol utilizando el objeto nodo creado en el ejercicio E-3.
NOTA: Los paréntesis no tienen que formar parte del árbol, sino que cada subárbol (que no sea una hoja del árbol) se encierra por un paréntesis para representar el orden de las operaciones.

Solución:




```
public String getExpression(BinaryNode<E> n){
        String s = "";
        if(n.leftChild==null || n.rightChild==null){
                s = n.value.toString();
                return s;
        }
        else{
                s = s.concat("(");
                s = s.concat(getExpression(n.leftChild));
                s = s.concat(n.value.toString());
                s = s.concat(getExpression(n.rightChild));
                s = s.concat(")");
                return s;
        }
}
```

3.6

Implemente en java las siguientes operaciones a realizar sobre un árbol binario:

a) preorderNext(v): devuelve el siguiente nodo visitado después de v según un recorrido preorden de un árbol.
b) inorderNext(v): devuelve el siguiente nodo visitado después de v según un recorrido inorden de un árbol.
c) postorderNext(v): devuelve el siguiente nodo visitado después de v según un recorrido postorden de un árbol.

¿Cuál es el peor caso considerando tiempos de ejecución de estos algoritmos?

Solución:

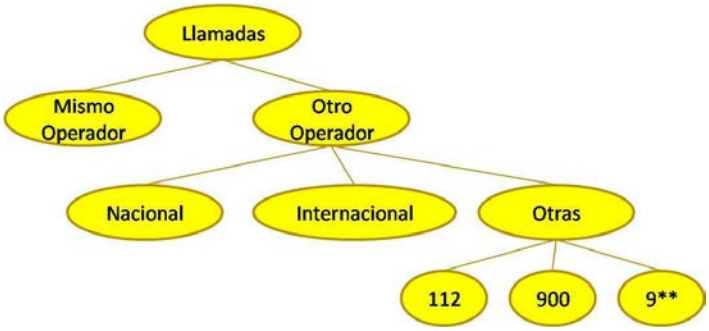
SOLUCIÓN:

```
/**
 * Exercise a
 */
public BinaryNode<E> preorderNext(BinaryNode<E> v){
	if(v.leftChild!=null){
		return v.leftChild;
	}
	else if(v.rightChild!=null){
		return v.rightChild;
	}
	else{
		BinaryNode<E> auxPadre = v.parent;
		BinaryNode<E> auxHijo = v;
		while(auxPadre.rightChild.equals(auxHijo)){
			auxHijo = auxPadre;
			auxPadre = auxHijo.parent;
		}
		return auxPadre.rightChild;
	}
}

/**
 * Exercise b
 */
public BinaryNode<E> inorderNext(BinaryNode<E> v){
	if(v.rightChild!=null){
		BinaryNode<E> aux = v.rightChild;
		while(aux.leftChild!=null){
			aux = aux.leftChild;
		}
		return aux;
	}
	else{
		if(v.parent.leftChild!=null &&
v.parent.leftChild.equals(v)){
			return v.parent;
		}
		else{
			BinaryNode<E> auxPadre = v.parent;
```

```
                    BinaryNode<E> auxHijo = v;
                    while(auxPadre.rightChild.equals(auxHijo)){
                          auxHijo = auxPadre;
                          auxPadre = auxHijo.parent;
                    }
                    return auxPadre;
              }
        }
}

/**
 * Exercise c
 */
public BinaryNode<E> postorderNext(BinaryNode<E> v){
        if(v.parent==null){
              return null;
        }
        if(v.parent.rightChild!=null &&
v.parent.rightChild.equals(v)){
              return v.parent;
        }
        else{
              if(v.parent.rightChild==null){
                    return v.parent;
              }
              else{
                    BinaryNode<E> aux = v.parent.rightChild;
                    while(aux.isInternal()){
                          if(aux.leftChild!=null){
                                aux = aux.leftChild;
                          }
                          else{
                                aux = aux.rightChild;
                          }
                    }
                    return aux;
              }
        }
}
```

3.7

A continuación se muestra un árbol genérico y la 'representación indentada' que resulta de él.
NOTA: Los hijos de cada nodo están incluidos entre paréntesis e indentados (una unidad más).



```
Llamadas(
	Mismo Operador
	Otro Operador(
		Nacional
		Internacional
		Otras(
			112
			900
			9**
		)
	)
)
```

Implementa en java el método que imprime por pantalla la representación indentada de un árbol genérico.

Solución:

```
public void represent(GenericNode<E> n,String indent){
	System.out.print(indent + n.value);
	if(n.hasChilds()){
		System.out.print("\n");
		System.out.print("\n");
		for(int i=0;i<n.childs.size();i++){
			represent(n.childs.get(i),indent + "\t");
		}
		System.out.print(indent + ")");
		System.out.print("\n");
	}
	else{
		System.out.print("\n");
	}
}
```

3.8

El factor de balance (Balance Factor) de un nodo interno v de un árbol binario es la diferencia entre las alturas de sus subárboles izquierdo y derecho. Describa un método para imprimir los factores de balanceo de todos los nodos internos de un árbol binario utilizando un recorrido in-orden.

Solución:

```
public void getFactors(BinaryNode<E> n){
	if(n==null){
		return;
	}
	if(n.leftChild!=null){
		getFactors(n.leftChild);
	}
	System.out.println(getHeight(n.leftChild)-
getHeight(n.rightChild));
	if(n.rightChild!=null){
		getFactors(n.rightChild);
```

```
	}
}
/**
 * Auxiliar Method used in Exercise E-8
 */
public int getHeight(BinaryNode<E> n){
	if(n==null){
		return 0;
	}
	if(n.leftChild==null && n.rightChild==null){
		return 1;
	}
	int height = 0;
	int leftHeight = getHeight(n.leftChild);
	int rightHeight = getHeight(n.rightChild);
	height = (leftHeight>rightHeight ? leftHeight :
rightHeight);
	height ++;
	return height;
}
```

3.9

Escriba un algoritmo que reciba como entrada la raíz de un árbol binario y que imprima el valor de los nodos en orden de nivel. El primer nivel imprime la raíz, después imprime todos los nodos de nivel 1, entonces los nodos de nivel 2 y así hasta el final.
NOTA: Recorridos pre-orden utilizan pilas para hacer llamadas recursivas. Considere utilizar estructuras de datos auxiliares para ayudarte a imprimir el recorrido por nivel.

Solución:

```
public void printTreeByLevel(BinaryNode<E> root){
	List<BinaryNode<E>> nodesToWalk=new
LinkedList<BinaryNode<E>>();
	nodesToWalk.add(root);
	List<BinaryNode<E>> nodesAux = new
LinkedList<BinaryNode<E>>();
	while(!nodesToWalk.isEmpty()){
		nodesAux = new LinkedList<BinaryNode<E>>();
		for(int i=0;i<nodesToWalk.size();i++){
			BinaryNode<E> nb = nodesToWalk.get(i);
			System.out.print(nb.value.toString() + " ");
			if(nb.hasLeft()){
				nodesAux.add(nb.leftChild);
			}
			if(nb.hasRight()){
				nodesAux.add(nb.rightChild);
			}
		}
		nodesToWalk = nodesAux;
		System.out.println("");
	}
}
```

3.10

Tenemos un árbol binario que almacena números enteros (integer) en sus nodos. Escriba (en código java) un método recursivo que sume todos los valores de los nodos del árbol.

Solución:

```
public int getAddition(BinaryNode<Integer> v){
	if(v.leftChild==null && v.rightChild==null){
		return v.value;
	}
	else if(v.leftChild==null){
		return v.value+getAddition(v.rightChild);
	}
	else if(v.rightChild==null){
		return v.value+getAddition(v.leftChild);
	}
	else{
		return v.value + getAddition(v.leftChild) +
getAddition(v.rightChild);
	}
}
```

3.11

Tenemos un árbol binario que almacena números enteros (integer) en sus nodos. Escriba un método recursivo que recorra el árbol e imprima los valores de cada nodo cuyo abuelo tenga un valor múltiplo de cinco.

Solución:

```
public void print5Grandparent(BinaryNode<Integer> v){
	if(v.parent!=null && v.parent.parent!=null &&
(v.parent.parent.value%5==0)){
		System.out.println(v.value);
	}
	if(v.leftChild!=null){
		print5Grandparent(v.leftChild);
	}
	if(v.rightChild!=null){
		print5Grandparent(v.rightChild);
	}
}
```

3.12

Implementa un método en java que calcule la media aritmética de las profundidades de todos los nodos hoja (externos) de un árbol binario.

Solución:

```
public float getAverageDepth(){
	int depths = 0;
	int nodes = 0;
	BinaryNode<E> aux = root;
	while(aux!=null){
		if(aux.isExternal()){
			depths +=getDepth(aux);
			nodes += nodes;
		}
		aux = preorderNext(aux);
	}
	return ((float)depths/(float)nodes);
}
```

3.13

Tenemos un árbol binario que almacena String con los nombres de los empleados de una empresa. Los diferentes niveles del árbol (profundidades) representan las categorías de los empleados.
Implementa un método en java que a la entrada recibe un nombre y una categoría y devuelve el número de empleados que se llaman así y pertenecen a esa categoría.

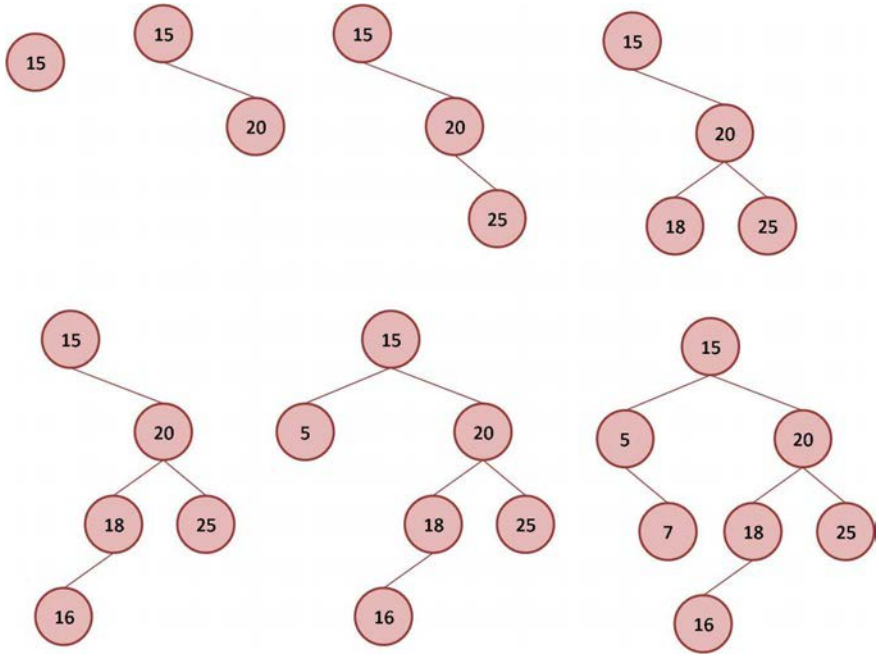
Solución:

```
public int getNumberNamedEmployees(String name, int category){
	int number = 0;
	BinaryNode<String> aux = root;
	while(aux!=null){
		if(getDepth(aux)==category && aux.value.equals(name)){
			number++;
		}
		aux = preorderNext(aux);
	}
	return number;
}
```

3.14

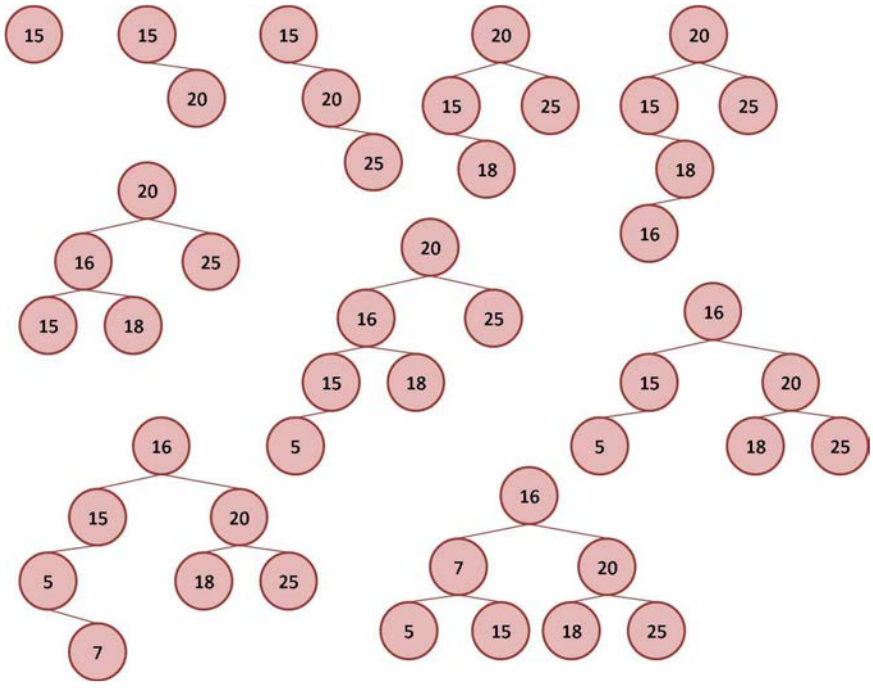
a) Dibuje el ABB resultante de insertar los valores 15, 20, 25, 18, 16, 5 y 7 en ese orden. Dibuje también los árboles intermedios después de cada inserción.
b) Escriba los recorridos pre-orden, in-orden y post-orden del árbol resultante de a).
c) Dibuje el ABB completamente equilibrado resultante de insertar los valores 15, 20, 25, 18, 16, 5 y 7 en ese orden. Dibuje también los árboles intermedios después de cada inserción.
d) Escriba los recorridos pre-orden, in-orden y post-orden del árbol resultante de c).

Solución:



a)

b) Pre-orden: 15-5-7-20-18-16-25
In-orden: 5-7-15-16-18-20-25
Post-orden: 7-5-16-18-25-20-15



c)

d) Pre-orden: 16-7-5-15-20-18-25
In-orden: 5-7-15-16-18-20-25
Post-orden: 5-15-7-18-25-20-16

3.15

Un alumno quiere almacenar sus ejercicios de EDA pero no los está resolviendo en orden. Por eso quiere utilizar para ello un árbol binario de búsqueda. De cada Ejercicio quiere almacenar el **nombre** (que estará compuesto por el número de la hoja y el número del ejercicio, por ejemplo, el ejercicio 4 de la hoja 1 sería 104), el **enunciado** y la **solución** (suponiendo que la solución es un String). Implementa en Java el árbol binario de Búsquedas que almacene los ejercicios e implementa los siguientes métodos:

a) Reemplazar un ejercicio: el alumno ha modificado un ejercicio ya existente y quiere reemplazar el antiguo.
b) Incluir un nuevo ejercicio.
c) Consultar un ejercicio en base a su nombre (identificador)
d) Eliminar un ejercicio.
e) Consultar cuantos ejercicios tiene hechos hasta el momento.

Solución:

```
public class Exercise {
	private String name;
	private String enunciado;
	private String solution;
	public Exercise() {
		super();
	}
public Exercise(String name, String enunciado, String solution) {
		super();
		this.name = name;
		this.enunciado = enunciado;
		this.solution = solution;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEnunciado() {
		return enunciado;
	}
	public void setEnunciado(String enunciado) {
		this.enunciado = enunciado;
	}
	public String getSolution() {
		return solution;
	}
	public void setSolution(String solution) {
		this.solution = solution;
	}
	public String toString(){
	return "[ " + name + "\t" + enunciado + "\t" + solution + "
]";
	}
}
```

```
////////////////////////////////////////////////////////////////

public class BinarySearchNode<K extends Comparable,V> {

	protected BinarySearchNode<K,V> parent;
	protected BinarySearchNode<K,V> leftChild;
	protected BinarySearchNode<K,V> rightChild;
	protected V value;
	protected K key;

	public BinarySearchNode(K k,V x){
		parent=null;
		leftChild=null;
		rightChild=null;
		value=x;
		key = k;
	}

	public BinarySearchNode(BinarySearchNode<K,V> padre,
BinarySearchNode<K,V> hijoIzquierda,
			BinarySearchNode<K,V> hijoDerecha, V valor, K
key) {
		super();
		this.parent = padre;
		this.leftChild = hijoIzquierda;
		this.rightChild = hijoDerecha;
		this.value = valor;
		this.key = key;
	}
	public BinarySearchNode<K,V> getPadre() {
		return parent;
	}
	public void setPadre(BinarySearchNode<K,V> padre) {
		this.parent = padre;
	}
	public BinarySearchNode<K,V> getHijoIzquierda() {
		return leftChild;
	}
	public void setHijoIzquierda(BinarySearchNode<K,V>
hijoIzquierda) {
		this.leftChild = hijoIzquierda;
	}
	public BinarySearchNode<K,V> getHijoDerecha() {
		return rightChild;
	}
	public void setHijoDerecha(BinarySearchNode<K,V>
hijoDerecha) {
		this.rightChild = hijoDerecha;
	}
	public V getValor() {
		return value;
	}
	public void setValor(V valor) {
		this.value = valor;
	}

	public K getKey() {
		return key;
	}
```

```
	public void setKey(K key) {
		this.key = key;
	}

	public BinarySearchNode<K,V> left(){
		return leftChild;
	}
	public BinarySearchNode<K,V> right(){
		return rightChild;
	}
	public boolean hasLeft(){
		if(leftChild!=null){
			return true;
		}
		return false;
	}
	public boolean hasRight(){
		if(rightChild!=null){
			return true;
		}
		return false;
	}
	public BinarySearchNode<K,V> parent(){
		return parent;
	}
	public boolean isInternal(){
		if(leftChild==null && rightChild==null){
			return false;
		}
		return true;
	}
	public boolean isExternal(){
		if(leftChild==null && rightChild==null){
			return true;
		}
		return false;
	}
	public BinarySearchNode<K,V> root(){
		if(parent==null){
			return this;
		}
		else{
			return parent.root();
		}
	}
	public boolean isRoot(){
		if(parent==null){
			return true;
		}
		else{
			return false;
		}
	}

}
```

```
////////////////////////////////////////////////////////////////

public class BinarySearchTree<K extends Comparable,V>{
	BinarySearchNode<K,V> root;
	int size;
	public int size(){
		return size;
	}
	public BinarySearchNode<K, V> root(){
		return root;
	}
	public BinarySearchTree(){
		root = null;
		size = 0;
	}
	public BinarySearchTree(BinarySearchNode<K,V> node){
		root = node;
		size = 1;
	}
	public void insertNode(BinarySearchNode<K,V> v){
		if(size==0){
			root = v;
			size++;
			return;
		}
		insertNode(v, root);
	}

	public void insertNode(BinarySearchNode<K,V>
n,BinarySearchNode<K,V> d){
		K key = n.key;
		if(key.compareTo(d.key)==0){
		System.out.println("Ya hay un elemento con esta clave,
no podemos a?adir otro");
			return;
		}
		else if(key.compareTo(d.key)<0){
			if(d.leftChild==null){
				d.leftChild = n;
				n.parent = d;
				size++;
				return;
			}
			else{
				insertNode(n, d.leftChild);
			}
		}
		else{
			if(d.rightChild==null){
				d.rightChild = n;
				n.parent = d;
				size++;
				return;
			}
			else{
				insertNode(n, d.rightChild);
			}
		}
	}
```

```
public BinarySearchNode<K,V> searchNode (BinarySearchNode<K,V> v,K
key){
		if(v==null){
			return null;
		}
		if(v.key.compareTo(key)==0){
			return v;
		}
		else{
			if(v.key.compareTo(key)<0){
				return searchNode(v.rightChild,key);
			}
			else{
				return searchNode(v.leftChild,key);
			}
		}
	}
	public BinarySearchNode<K,V> removeNode(K key){
		if(size==0){
			System.out.println("La lista est vac a, no
podemos eliminar el nodo deseado");
			return null;
		}
		return removeNode(key,root,0);
	}
	public BinarySearchNode<K,V> removeNode(K key,
BinarySearchNode<K,V> node, int lado){
		BinarySearchNode<K,V> aux = node;
		if(key.equals(node.key)){
			if(node.isExternal()){
				if(node.isRoot()){
					root = null;
				}
				else{
					if(lado==1){
						aux.parent.leftChild = null;
					}
					else {
						aux.parent.rightChild = null;
					}
				}
			}
			else if(aux.leftChild==null){
				if(aux.isRoot()){
					root = aux.rightChild;
				}
				else{
					if(lado==1){
				aux.parent.leftChild = aux.rightChild;
				aux.rightChild.parent = aux.parent;
					}
					else {
				aux.parent.rightChild = aux.rightChild;
				aux.rightChild.parent = aux.parent;
					}
				}
			}
			else if(node.rightChild==null){
```

```
                        if(aux.isRoot()){
                              root = aux.leftChild;
                        }
                        else{
                              if(lado==1){
                        aux.parent.leftChild = aux.leftChild;
                        aux.leftChild.parent = aux.parent;
                              }
                              else {
                        aux.parent.rightChild = aux.leftChild;
                        aux.leftChild.parent = aux.parent;
                              }
                        }
                  }
                  else{
                  BinarySearchNode<K,V> aux2 = aux.rightChild;
                        while(aux2.leftChild!=null){
                              aux2 = aux2.leftChild;
                        }
                        if(aux2.parent.key==aux.key){
                        aux2.parent.rightChild = aux2.rightChild;
                        }
                        else{
                        aux2.parent.leftChild = aux2.rightChild;
                        }
                        aux2.leftChild = aux.leftChild;
                        if(aux.leftChild!=null){
                              aux.leftChild.parent = aux2;
                        }
                        aux2.rightChild = aux.rightChild;
                        if(aux.rightChild!=null){
                              aux.rightChild.parent = aux2;
                        }
                        if(aux.isRoot()){
                              root = aux2;
                        }
                        else{
                              if(lado==1){
                                    aux.parent.leftChild = aux2;
                                    aux2.parent = aux.parent;
                              }
                              else {
                                    aux.parent.rightChild = aux2;
                                    aux2.parent = aux.parent;
                              }
                        }
                  }
                  size--;
                  return aux;
            }
            else if(key.compareTo(aux.key)<0){
                  if(aux.leftChild==null){
                  System.out.println("No existe la clave, as? que
no eliminamos nada");
                        return null;
                  }
                  else{
                        return removeNode(key,node.leftChild,1);
                  }
```

```
			}
			else{
				if(aux.rightChild==null){
					System.out.println("No existe la clave,
as\ufffd que no eliminamos nada");
					return null;
				}
				else{
					return removeNode(key,node.rightChild,2);
				}
			}
		}

	public BinarySearchTree<Integer,Integer> generateRandomTree(int
n){
			BinarySearchTree<Integer,Integer> tree = new
BinarySearchTree<Integer,Integer>();
			boolean[] libres = new boolean[n];
			for(int i=0;i<libres.length;i++){
				libres[i] = true;
			}
			while(stillFree(libres)){
				int number = (int)(Math.random()*n)+1;
				System.out.println("1Buscamos el "+number);
				while(!libres[number-1]){
					number = (int)(Math.random()*n)+1;
					System.out.println("Buscamos el "+number);
				}
				libres[number-1] = false;
				BinarySearchNode<Integer,Integer> node = new
BinarySearchNode<Integer,Integer>(number, number);
				tree.insertNode(node);
			}
			return tree;
		}

	private boolean stillFree(boolean[] in){
			for(int i=0;i<in.length;i++){
				if(in[i]==true){
					return true;
				}
			}
			return false;
		}
		public String toString(){
			String s = "";
			s = s.concat(toString(root,""));
			return s;
		}
	public String toString(BinarySearchNode<K,V> node,String aux){
			if(node==null){
				return "";
			}
			String s = "";
			s = s.concat(node.getKey() + "\n");
			if(node.leftChild!=null){
				s = s.concat(aux + "\t" + "i " +
toString(node.leftChild,aux + "\t"));
			}
```

```
            if(node.rightChild!=null){
                  s = s.concat(aux + "\t" + "d " +
toString(node.rightChild,aux + "\t"));
            }
            return s;
      }


public BinarySearchNode<K,V> preorderNext(BinarySearchNode<K,V>
v){
            if(v.leftChild!=null){
                  return v.leftChild;
            }
            else if(v.rightChild!=null){
                  return v.rightChild;
            }
            else{
                  BinarySearchNode<K,V> auxPadre = v.parent;
                  BinarySearchNode<K,V> auxHijo = v;
                  while(auxPadre!=null){
                        if(auxPadre.rightChild==null ||
auxPadre.rightChild.equals(auxHijo)){

      //while(auxPadre.rightChild.equals(auxHijo)){
                              auxHijo = auxPadre;
                              auxPadre = auxPadre.parent;
                              if(auxPadre==null){
                                    return null;
                              }
                        }
                        else{
                              break;
                        }
                  }
                  //Llegamos al root
                  return auxPadre.rightChild;
            }
      }

public BinarySearchNode<K,V> inorderNext(BinarySearchNode<K,V> v){
            if(v.rightChild!=null){
                  BinarySearchNode<K,V> aux = v.rightChild;
//Buscamos el que baje hasta abajo en el siguiente nodo si es que
hay otro, no??
                  while(aux.leftChild!=null){
                        aux = aux.leftChild;
                  }
                  return aux;
            }
            else{
                  if(v.parent.leftChild!=null &&
v.parent.leftChild.equals(v)){
                        return v.parent;
                  }
                  else{
                        BinarySearchNode<K,V> auxPadre = v.parent;
                        BinarySearchNode<K,V> auxHijo = v;
                        while(auxPadre!=null){
```

```
                              if(auxPadre.rightChild==null ||
auxPadre.rightChild.equals(auxHijo)){
                              auxHijo = auxPadre;
                                    auxPadre = auxPadre.parent;
                                    if(auxPadre==null){
                                          return null;
                                    }
                              }
                              else{
                                    return auxPadre;//break;
                              }
                        }
                        return auxPadre;
                  }
            }
      }

public BinarySearchNode<K,V> postorderNext(BinarySearchNode<K,V>
v){
            if(v.parent==null){
                  return null;
            }
            if(v.parent.rightChild!=null &&
v.parent.rightChild.equals(v)){
                  return v.parent;
            }
            else{
                  if(v.parent.rightChild==null){
                        return v.parent;
                  }
                  else{
                        //Hay que buscar el hijo que toca
                        //Buscamos el que m s a la izquierda est
                  BinarySearchNode<K,V> aux = v.parent.rightChild;
                        while(aux.isInternal()){
                              if(aux.leftChild!=null){
                                    aux = aux.leftChild;
                              }
                              else{
                                    aux = aux.rightChild;
                              }
                        }
                        return aux;
                  }
            }
      }

}

////////////////////////////////////////////////////////////////

public class ExerciseBST extends
BinarySearchTree<Integer,Exercise>{
      public void insertExercise(Exercise e){
            BinarySearchNode<Integer, Exercise> v = new
BinarySearchNode<Integer, Exercise>(new Integer(e.getName()), e);
            this.insertNode(v);
      }
      public Exercise getExercise(String name){
```

```
            BinarySearchNode<Integer, Exercise> v =
this.searchNode(this.root(), new Integer(name));
            if(v!=null){
                  return v.getValor();
            }
            return null;
      }
      public Exercise replaceExercise(Exercise o, Exercise n) {
            BinarySearchNode<Integer, Exercise> v =
this.searchNode(this.root(), new Integer(o.getName()));
            Exercise result = null;
            if(v!=null){
                  result = v.getValor();
                  v.setValor(n);
                  return result;
            }
            return null;
      }
      public Exercise deleteExercise(String name){
            BinarySearchNode<Integer, Exercise> v =
this.removeNode(new Integer(name));
            if(v!=null){
                  return v.getValor();
            }
            return null;
      }
      public void mostrarPreOrder(){
            BinarySearchNode<Integer, Exercise> aux = this.root();
            while(aux!=null){
                  System.out.println(aux.getValor().toString());
                  aux = this.preorderNext(aux);
            }
            System.out.println("---------------");
      }
      public void mostrarInOrder(){
            BinarySearchNode<Integer, Exercise> aux = this.root();
            while(aux.left()!=null){
                  aux = aux.left();
            }
            while(aux!=null){
                  System.out.println(aux.getValor().toString());
                  aux = this.inorderNext(aux);
            }
            System.out.println("---------------");
      }
      public void mostrarPostOrder(){
            BinarySearchNode<Integer, Exercise> aux = this.root();
            while(aux.left()!=null){
                  aux = aux.left();
            }
            while(aux!=null){
                  System.out.println(aux.getValor().toString());
                  aux = this.postorderNext(aux);
            }
            System.out.println("---------------");
      }
}
```

```
////////////////////////////////////////////////////////////////

public class Student {
	ExerciseBST exercises = new ExerciseBST();
	/**
	 * Reemplazar un ejercicio: el alumno ha modificado un
ejercicio ya existente y quiere reemplazar el antiguo.
	 */
	public Exercise replaceExercise(Exercise o, Exercise n){
		if(!o.getName().equals(n.getName())){
			System.out.println("They are not the same
exercise. Can not replace it.");
			return null;
		}
		return exercises.replaceExercise(o,n);
	}
	/**
	 * Incluir un nuevo ejercicio.
	 */
	public void addExercise(Exercise n){
		exercises.insertExercise(n);
	}
	/**
	 * Eliminar un ejercicio.
	 */
	public Exercise deleteExercise(String name){
		return exercises.deleteExercise(name);
	}
	/**
	 * Consultar un ejercicio en base a su nombre
(identificador)
	 */
	public Exercise searchExercise(String name){
		return exercises.getExercise(name);
	}
	/**
	 * Consultar cuantos ejercicios tiene hechos hasta el
momento.
	 */
	public int listNumberExercises(){
		return exercises.size();
	}
	public void printExercises(){
		exercises.mostrarInOrder();
	}
	public static void main (String args[]){
	Student s1 = new Student();
	Exercise e1 = new Exercise("101", "enun101", "sol101");
	Exercise e2 = new Exercise("107", "enun107", "sol107");
	Exercise e3 = new Exercise("112", "enun112", "sol112");
	Exercise e4 = new Exercise("205", "enun205", "sol205");
	Exercise e5 = new Exercise("220", "enun220", "sol220");
	Exercise e6 = new Exercise("206", "enun206", "sol206");
		s1.addExercise(e3);
		s1.addExercise(e6);
		s1.addExercise(e2);
		s1.addExercise(e1);
		s1.addExercise(e5);
		s1.addExercise(e4);
```

```
            s1.printExercises();
            Exercise ee = s1.replaceExercise(e2, new
Exercise("107","enun107_2","sol107_2"));
            if(ee!=null){
                  s1.printExercises();
            }
            Exercise ee2 = s1.replaceExercise(e3, new
Exercise("113","enun113_2","sol113_2"));
            if(ee2!=null){
                  s1.printExercises();
            }
            s1.deleteExercise("206");
            s1.printExercises();
      }
}
```

3.16

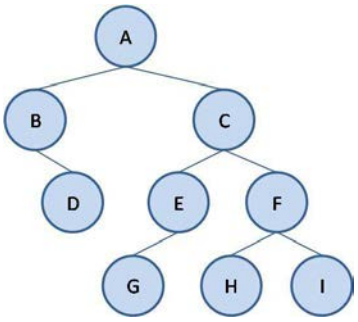


Figura E-20-1

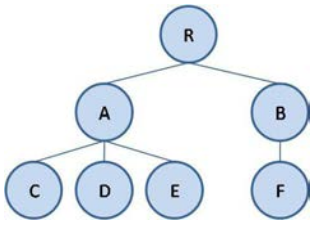


Figura E-20-2

a) Dibuje el árbol binario resultante de la representación de la siguiente secuencia suponiendo que la representación del árbol de la figura E-20-1 es AB/D//CEG///FH//I// donde cada / representa un nodo nulo.

ABD//E//C/F//

a) Dibuje el árbol binario resultante de la representación de la siguiente secuencia suponiendo que la representación del árbol de la figura E-20-1 es A'B'/DC'E'G/F'HI donde cada / representa un nodo nulo y cada ' representa que un nodo tiene al menos un descendiente.

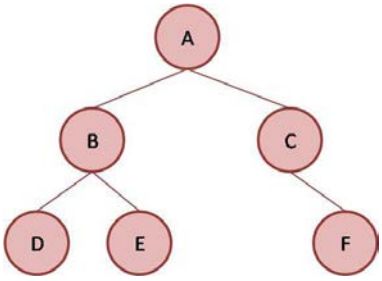
A'/B'/C'D'G/E

b) Dibuje el árbol GENERAL resultante de la representación de la siguiente secuencia suponiendo que la representación del árbol de la figura E-20-2 es RAC)D)E))BF))) donde cada) representa un nodo no tiene más descendientes.

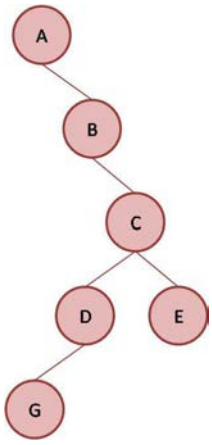
XPC)Q)RV)M))))

Solución:

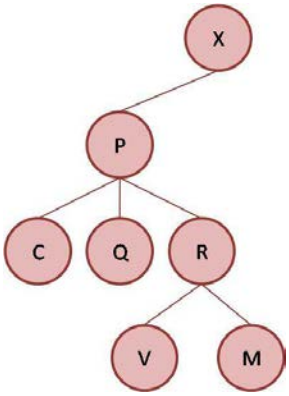
a)



b)



c)

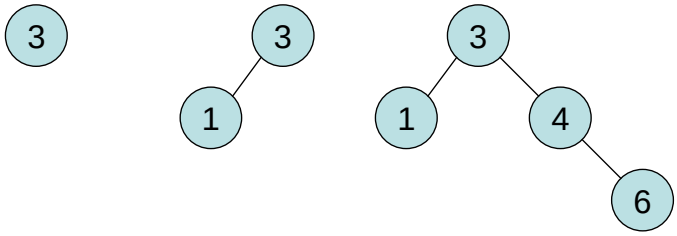


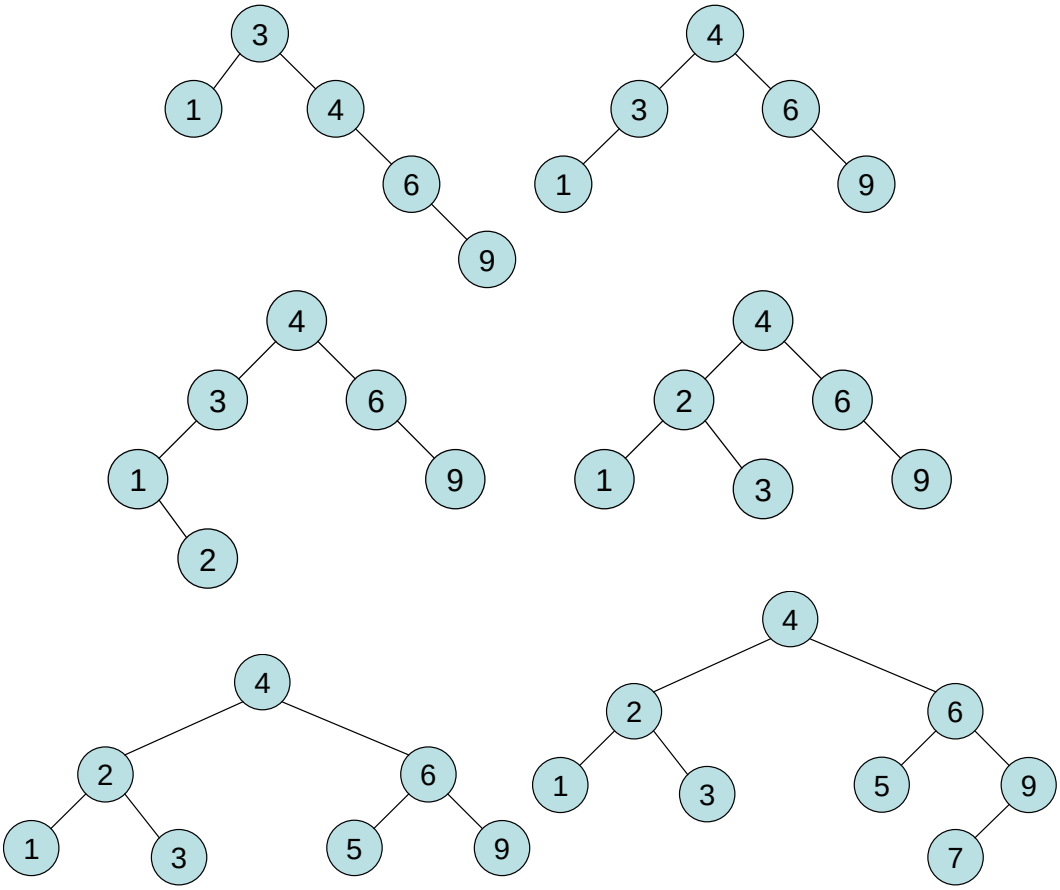
3.17

a) Muestre todos los resultados de insertar 3, 1, 4, 6, 9, 2, 5, 7 en un ABB equilibrado completamente vacío al inicio.
b) Muestre el resultado de eliminar la raíz del mismo árbol.

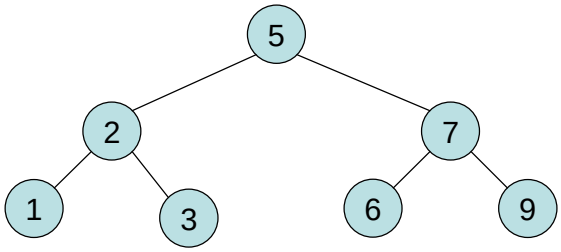
Solución:

a)





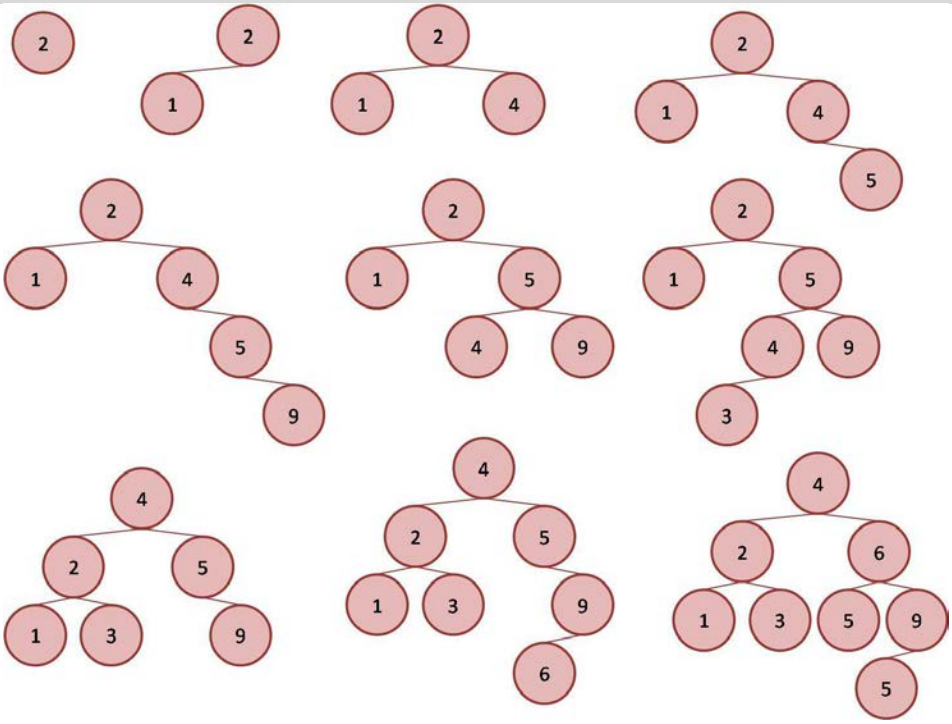
b)



3.18

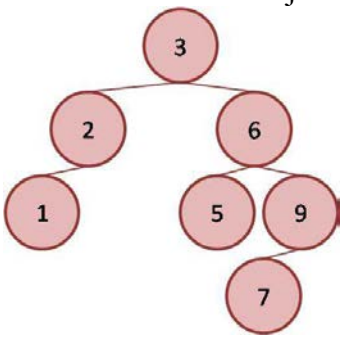
a) Muestre todos los resultados intermedios de insertar 2, 1, 4, 5, 9, 3, 6, 7 en un árbol AVL inicialmente vacío.
b) Elimine la raíz del árbol y muestre el resultado.

Solución:

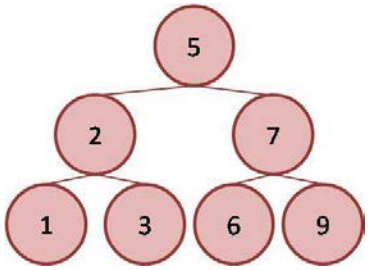


a)

b) Tenemos dos opciones de solucionar este ejercicio.



- si se coge el mayor de los descendientes de la izqueirda.



- si se coge el menor de los descendientes de la derecha.

3.19

Escribe un método en java que genere un árbol binario de búsqueda aleatorio de n nodos donde los nodos tengan como clave números desde 1 hasta n.

Solución 1:

Utilizando un array auxiliar para ver si la clave ya está usada.

```
public BinarySearchTree<Integer> generateRandomTree(int n){
	BinarySearchTree<Integer> tree =new
BinarySearchTree<Integer>();
	boolean[] libres = new boolean[n];
	for(int i=0;i<libres.length;i++){
		libres[i] = true;
	}
	while(stillFree(libres)){
		int number = (int)(Math.random()*n)+1;
		while(!libres[number-1]){
			number = (int)(Math.random()*n)+1;
		}
		libres[number-1] = false;
		BinarySearchNode<Integer> node =new
				BinarySearchNode<Integer>(number,
number);
		tree.insertNode(node);
	}
	return tree;
}
private boolean stillFree(boolean[] in){
	for(int i=0;i<in.length;i++){
		if(in[i]==true){
			return true;
		}
	}
	return false;
}
```

Solución 2:

Comprobando si la clave ya está incluída en el árbol.

```
public BinarySearchTree<Integer,Integer> generateRandomTree2(int
n){
	BinarySearchTree<Integer,Integer> tree = new
BinarySearchTree<Integer,Integer>();
	while(tree.size<n){
		int number = (int)(Math.random()*n)+1;
		while(tree.searchNode(tree.root, new
Integer(number))!=null){
			number = (int)(Math.random()*n)+1;
		}
		BinarySearchNode<Integer,Integer> node = new
BinarySearchNode<Integer,Integer>(number, number);
		tree.insertNode(node);
	}
```

```
      return tree;
}
```

3.20

Dada la siguiente implementación de un árbol general

```
import java.util.ArrayList;
/**Clase que implementa un árbol general*/
public class Tree<E> {
      /**devuelve nº. de nodos del árbol */
      public int size();
      /**devuelve true si el árbol es vacio, false en otro caso */
      public boolean isEmpty();
        /**devuelve la raíz del árbol; null si es vacio */
      public NodeTree<E> getRoot();
      /**devuelve el padre de v*/
      public NodeTree<E> parent(NodeTree<E> v);
      /**devuelve un arraylist que contiene los hijos del nodo v*/
      public ArrayList<NodeTree<E>> children(NodeTree<E> v);
      /**testea si el nodo es interno.*/
      public boolean isInternal(NodeTree<E> v);
      /**testea si el nodo es externo.*/
      public boolean isLeaf(NodeTree<E> v);
      /**testea si el nodo es la raiz.*/
      public boolean isRoot(NodeTree<E> v);
                                          …
}
```

Escribe un método **iterativo**, **getListNodes()**, que devuelve en un arraylist todos los nodos del árbol.

Solución:

```
public ArrayList<NodeTree<E>> getListNodesIte() {
            ArrayList<NodeTree<E>> lstNodes = new
ArrayList<NodeTree<E>>();
            if (isEmpty()) return lstNodes;
            //add the root
            lstNodes.add(root);
            //we should traverse each node of lstNodes and
            // add its children at the last of the list (please,
note that lstNodes may be increased
            // by adding the children)
            NodeTree<E> child;
            for(int i=0; i<lstNodes.size(); i++){
                  child = lstNodes.get(i);
                  ArrayList<NodeTree<E>> list= children(child);
                  if (list != null){
                        for(NodeTree<E> c:list){
                              lstNodes.add(c);
                        }
                  }
            }
            return lstNodes;
      }
```

3.21

Escribe un método recursivo que dado un nodo de un árbol binario de enteros calcule cuantos de sus descendientes son pares.

Solución:

```
public int getNumDescendentsPar(BinaryNode<E> n){
      if(n!=null) return 0;

      int numDesc= 0;
      if (n.item%2==0) numDesc++;

      if(hasLeft(n)){
            numDesc += calculateDescendents(n.left);
      }
      if(hasRight(n)){
            numDesc += calculateDescendents(n.right);
      }
      return numDesc;
}
```

3.22

Dado un árbol binario de String, escriba un método recursivo que muestre todos los valores de los nodos cuyo valor de su nodo abuelo empieza con la palabra “sol” en recorrido postorder. Te recomendamos utilizar el método booleano **startsWith** de la clase String. (Ejemplo, String str=”” str.startsWith("sol");)

Solución:

```
   public void printSolGrandparent(BinaryNode<String> v){

      if(hasLeft(v)){
            printSolGrandparent (v.leftChild);
      }
      if(hasRight(v)){
            printSolGrandparent (v.rightChild);
      }
      if(v.parent!=null && v.parent.parent!=null &&
      (v.parent.parent.value. startsWith(“sol”){
                  System.out.println(v.value);
            }

   }
```

3.23

Implementa el método postOrderNext en java. Este método recibe a la entrada un nodo (cualquiera) de un árbol binario y tiene que devolver el siguiente nodo del árbol en ser visitado en un recorrido postorden.

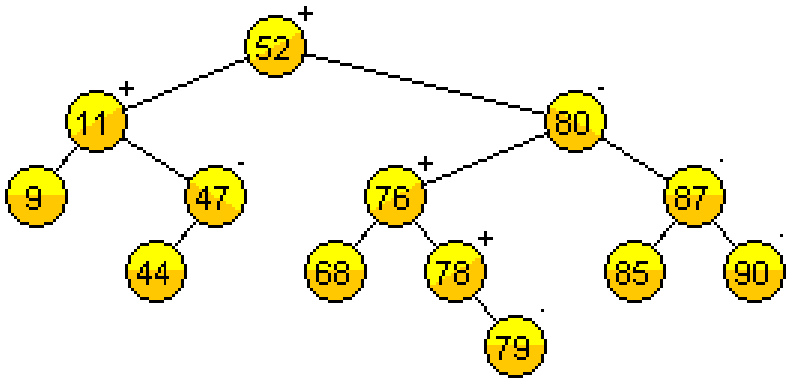
Solución:

```
public BinaryNode<E> postorderNext(BinaryNode<E> v){
	if(v.parent==null){
		return null;
	}
	if(v.parent.rightChild!=null &&
v.parent.rightChild.equals(v)){
		return v.parent;
	}
	else{
		if(v.parent.rightChild==null){
			return v.parent;
		}
		else{
			BinaryNode<E> aux = v.parent.rightChild;
			while(aux.isInternal()){
				if(aux.leftChild!=null){
					aux = aux.leftChild;
				}
				else{
					aux = aux.rightChild;
				}
			}
			return aux;
		}
	}
}
```

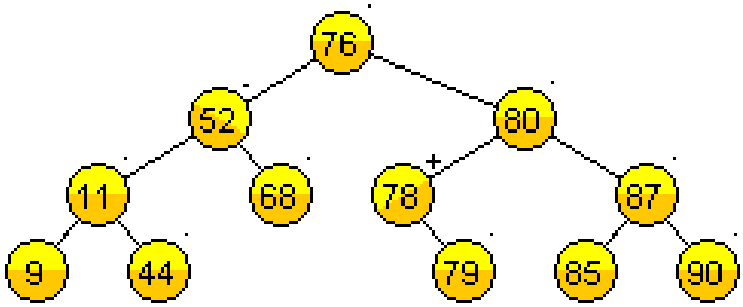
3.24

a) Muestre el resultado insertar 80, 9, 90, 52, 85, 68, 47, 78, 11, 87, 44, 76, 79 en un árbol AVL inicialmente vacío.
b) Elimina el nodo 47 del árbol y muestre el resultado.
c) Elimina la raíz y muestre el resultado.
d) Elimina de nuevo la raíz y muestre el resultado.

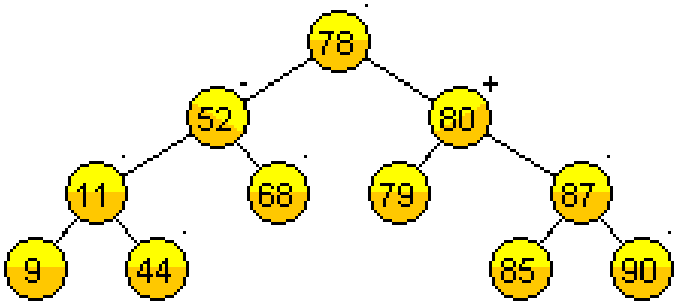
Solución:



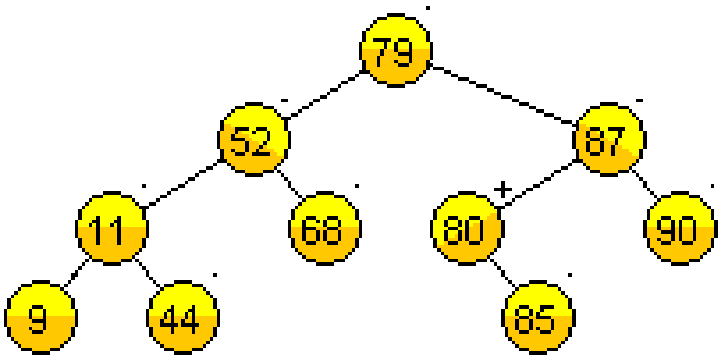
a)



b)



c)


79
52
87
11
68
80
90
9
44
85

Capitulo 4

Caso Práctico

Introducción

El propósito de esta práctica es desarrollar una aplicación para controlar las aulas informáticas de la Universidad.

La aplicación controlará el tiempo que los ordenadores están ocupados, de forma que pueda expulsar a los alumnos que han estado conectados más tiempo del permitido cuando hay otros estudiantes esperando.

Clases

La aplicación estará formada por clases de objetos para los estudiantes, aulas y ordenadores, las estructuras de datos maestras y operativas para almacenarlos, una clase de gestión y una de pruebas..

Clases de objetos

Alumno

La clase Alumno contiene el campo "id", que será asignado de forma automática y consecutiva, así como apellido y nombre.

```
public class Alumno {
  int id;
  String apellido;
  String nombre;
  public String toString () {
        return id + "\t" + apellido + ", " + nombre;
  }
}
```

Aula

La clase Aula contiene el campo nombre, con un formato del tipo 1.1.B10, así como un indicador de si el aula está cerrada, y la lista de ordenadores que contiene.

```
public class Aula {
  String nombre;
  boolean cerrada;
  ListaOrdenadores listaOrdenadores;
  public String toString () {
        return nombre;
  }
}
```

Ordenador
La clase Ordenador contiene el identificador completo, formada por el nombre del aula y in número consecutivo. Si el ordenador está ocupado, también almacena al alumno que lo ocupa y la hora que era cuando lo ocupó (el número de minutos transcurridos).

```
public class Ordenador {
  String nombreAula;
  int numero;
  Alumno alumno;                          // null si no está
ocupado
  int minutosTranscurridos;      // desde que fue ocupado
  public String toString () {
        return nombreAulda + "-" + new
DecimalFormat("00").format(numero);
  }
}
```

<u>Estructuras de datos maestras</u>
Se crearán las siguientes estructuras de datos para almacenar los datos maestros:

Alumnos por nombre (ArbolAlumnosPorNombre).
Loas alumnos se almacenarán en un árbol binario de búsqueda no balanceado, ordenados por nombre, esto es, por (apellido, nombre).

Alumnos por ID (ArbolAlumnosPorID).
En paralelo, los alumnos se almacenarán también en una lista ordenada, o en un árbol AVL, ordenados por ID.

Aulas informáticas (ListaAulas).
Las aulas se almacenarán en una lista, que podrá estar implementada a partir de una colección de Java.

Ordenadores en cada aula (ListaOrdenadores).
Los ordenadores en cada aula se almacenarán en una lista, que podrá estar implementada a partir de una colección de Java. No habrá una lista global para almacenar los ordenadores del campus.

Estructuras de datos operativas

Cola de alumnos (ColaAlumnos)

Se creará una clase que implemente la cola de estudiantes, que será una para todo el campus (no una por aula).

Ordenadores ocupados

No hay necesidad de crear ninguna estructura de datos para almacenar los ordenadores ocupados; es suficiente con la información que contiene la clase Ordenador.

Gestión y pruebas

Gestor

La clase Gestor, que tendrá una sola instancia, contendrá las estructuras de datos y controlará el resto de las clases.

También será responsable de la gestión del paso del tiempo, simulando el paso de los minutos transcurridos desde que la aplicación fue iniciada.

```
public class Gestor {
  private ArbolAlumnosPorNombre arbolAlumnosPorNombre = new
ArbolAlumnosPorNombre();
  private ListaAlumnosPorId listaAlumnosPorId = new
ListaAlumnosPorId();
  // o bien: private ArbolAlumnosPorId arbolAlumnosPorId = new
ArbolAlumnosPorId();
  private ListaAulas listaAulas  = new ListaAula();
  private ColaAlumnos colaAlumnos = new ColaAlumnos();
  private int minutosTranscurridos;
}
```

Estos campos son privados, de forma que no sean accesibles desde fuera.

Prueba

La clase estática Prueba contendrá diferentes métodos para probar las funcionalidades de las demás clases. También será responsable de rellenar los datos maestros, y contendrá el método main.

Operaciones sobre las estructuras de datos

PREGUNTAS – Todos los métodos descritos en esta sección DEBEN incluir un comentario en el que se indique su complejidad (1, log N, N, N log N, N^2, etc.), junto con una breve justificación y una indicación de lo que N significa en cada caso.

Las operaciones básicas sobre las estructuras de datos incluirán:

Lista de aulas

Esta clase estará basada en ArrayList, con un método adicional (El resto de funcionalidades podrán estar accesibles a través de ArrayList):

```
public class ListaAulas extends ArrayList<Aula> {
  public Aula obtenerPorNombre(String nombre) { ... }
}
```

Lista de ordenadores

Esta clase estará basada en ArrayList, sin añadir métodos adicionales (los presentes en ArrayList deberían ser suficientes):

```
public class ListaOrdenadores extends ArrayList<Ordenador> {
}
```

Árbol de alumnos por nombre

Esta clase no estará basada en estructuras de Java, y tendrá los siguientes métodos:

```
public class ArbolAlumnosPorNombre {
       public void añadir(Alumno alumno) { ... }
       public Alumno obtenerAlumno(String apellido, String
nombre) { ... }
       public ArrayList<Alumno>
obtenerListaAlumnosConApellido(String apellido) { ... }
}
```

Árbol / lista alumnos por ID

Esta clase no estará basada en estructuras de Java, y tendrá los siguientes métodos:

```
public class ListaAlumnosPorId {
       public void añadir(Alumno alumno) { ... }
       public Alumno obtenerAlumno(int id) { ... }
}
```

o bien:

```
public class ArbolAlumnosPorId {
       public void añadir(Alumno alumno) { ... }
       public Alumno obtenerAlumno(int id) { ... }
}
```

Recuerda que esta estructura deberá ser bien una lista ordenada o bien un árbol AVL.

PREGUNTA – Se debe responder aunque no se haya escogido la implementación como árbol:

¿Por qué hemos elegido implementar este árbol como AVL, en lugar del árbol por nombre?

Cola de alumnos

Esta clase no estará basada en estructuras de Java, y tendrá los siguientes métodos:

```
public class ColaAlumnos {
  public Alumno mirarPrimerAlumno() { ... }
  public Alumno desencolarPrimerAlumno() { ... }
  public void encolarAlumno(Alumno alumno) { ... }
  public Alumno eliminarAlumnoPorId(int id) { ... }
}
```

PREGUNTA – ¿Cuál de estos métodos no pertenece al tipo abstracto de datos de una cola?

Operaciones básicas de gestión

Operaciones simples sobre los datos maestros

El objeto Gestor contendrá los siguientes métodos, que operarán directamente sobre las estructuras de datos mediante una simple llamada a los mismos:

```
public class Gestor {
 public boolean puedeAñadirAlumno(int id,String
apellido,String nombre) { ... }
 public void añadirAlumno(int id,String apellido,String
nombre) { ... }
 public void añadirAula(String nombre,int numeroOrdenadores)
{ ... }
 public Alumno obtenerAlumnoPorId(int id) { ... }
 public Alumno obtenerAlumnoPorNombre(String apellido,String
nombre) { ... }
}
```

Estos métodos deberían tener un número reducido de líneas de código, por ejemplo:

```
public class Gestor {
 ListaAlumnosPorId listaAlumnosPorId = new
ListaAlumnosPorId();
 public Alumno obtenerAlumnoPorId(int id) {
       Alumno alumno =
this.listaAlumnosPorId.obtenerAlumno(id);
       return alumno;
 }
}
```

Más operaciones sobre datos maestros

El objeto Gestor tendrá algunas operaciones no tan simples sobre las estructuras de datos maestras:

```
public class Gestor {
 public List<Ordenador> obtenerOrdenadoresLibres() { ... }
 public List<Ordenador> obtenerOrdenadoresOcupados() { ... }
 public Ordenador obtenerOrdenadorOcupadoPorMasTiempo() { ...
}
 public ArrayList<Alumno> obtenerListaAlumnosTrabajando() {
... }
}
```

Sólo debería iterarse sobre los ordenadores que estén en aulas abiertas (un ordenador en un aula que no esté abierta no se considera que esté libre, y por supuesto tampoco puede estar ocupado).

Ocupación y liberación de ordenadores

El objeto Gestor dispondrá de operaciones sencillas que serán llamadas cuando un alumno ocupa un ordenador libre (normalmente después de pasar por la cola) o libera un ordenador (bien voluntariamente, o porque el sistema le fuerza por haber expirado su tiempo:

```
public class Gestor {
  private void ocuparOrdenador(Ordenador ordenador, Alumno
alumno) { ... }
  private Alumno liberarOrdenador(Ordenador ordenador) { ... }
}
```

Estos métodos deben ser privados porque no están pensados para ser llamados desde fuera (desde la clase Prueba), sino desde métodos propios de esta clase. Habrá otros métodos más adelante, a los que se llamará cuando, por ejemplo, un alumno quiera abandonar un ordenador.

Operaciones de gestión avanzadas

Estos métodos deben escribir en la salida estándar de cara a realizar un seguimiento de la evolución del sistema. Se da un ejemplo de traza al final de este documento.

Gestión del tiempo

Para simular el paso del tiempo, el objeto Gestor dispondrá de una variable minutosTranscurridos para controlar el número de minutos que han pasado, esta variable será incrementada de forma simulada por un método que se llamará avanzarMinutosTranscurridos.

También habrá un método, gestionarCola, que será llamado para comprobar si hay alumnos en la cola, y estos pueden ocupar un ordenador que esté libre o que pueda ser liberado:

```
public class Gestor {
  int minutosTranscurridos = 0;
  public void avanzarMinutosTranscurridos (int minutos) {
        while (minutos > 0) {
              ++this.minutosTranscurridos ;
              --minutos;
              gestionarCola();
        }
  }
  private void gestionarCola() {
        // mientras hay alumnos en la cola
        //   si no hay ordenadores libres, busca el ordenador
usado por más tiempo,
        //     si el alumno ha excedido el tiempo de uso,
liberar el ordenador
        //   si hay un ordenador libre
        //        ocuparlo por el primer alumno en la cola
        //   si no
        //     salir
  }
}
```

De nuevo gestionarCola debe ser privado, ya que es para uso interno. Debe ser llamado al final de ciertos métodos, como en este caso lo hace avanzarMinutosTranscurridos).

Cola de alumnos

Los alumnos pueden ponerse a la cola, o pueden abandonarla (o abandonar el ordenador, si es que ya están trabajando):

```
public class Gestor {
  public Alumno ponerAlumnoEnCola(int id) { ... }
  public Alumno quitarAlumnoDeColaUOrdenador(int id) { ... }
}
```

Cuando un alumno se pone a la cola o abandona un ordenador, será necesario llamar a gestionarCola para comprobar si hay ordenadores libres y alumnos que los ocupen.

Gestión de aulas

Finalmente, las aulas, que por defecto están abiertas, pueden ser cerradas y reabiertas.

```
public class Gestor {
  public void abrirAula(String nombreAula) { ... }
  public void cerrarAula(String nombreAula) { ... }
}
```

Cuando se cierra un aula, los ordenadores deben ser liberados, y los alumnos que los ocupan se ponen de nuevo a la cola. Y cuando se abre, es necesario llamar a gestionarCola.

Pruebas

Pruebas de las estructuras de datos

Cada estructura de datos deberá tener al menos un método que la pruebe. Por ejemplo, para probar ColaAlumnos:

```
public class Prueba {
  static void pruebaColaAlumnos1() {
        ColaAlumnos colaAlumnos = new ColaAlumnos();
        ...
        colaAlumnos.mostrar();
  }
  public static void main(String[] args) {
        pruebaColaAlumnos1();
  }
}
```

Rellenar las estructuras maestras

Antes de poder probar la clase Gestor, se debe rellenar sus estructuras de datos, y esta es una responsabilidad de la clase Prueba, que tendrá estos métodos:

```
public class Prueba {
  static void rellenarAulasYOrdenadores ( Gestor gestor ) {
... }
  static void rellenarAlumnos(Gestor gestor, int id1, int n) {
... }
}
```

y serán invocados de esta forma:

```
public class Prueba {
  static void pruebaGestor1() {
        Gestor gestor = new Gestor();
```

```
        rellenarAlumnos(gestor, 10001, 100);
        rellenarAulasYOrdenadores(gestor);
        ...
  }
}
```

Estas funciones realizarán llamadas a los métodos descritos anteriormente.

Aulas y ordenadores en cada aula (rellenarAulasYOrdenadores)
La lista de ordenadores tendrá al menos tres aulas.

La lista de ordenadores para cada aula se rellenará con un número diferente pero pequeño de ordenadores con números consecutivos. Habrá pocos ordenadores en cada aula (menos de cinco), de forma que se ocupen rápidamente y la información se pueda visualizar fácilmente.

Alumnos (rellenarAlumnos)
Habrá un método para rellenar los árboles de alumnos con al menos 100 alumnos con Ids consecutivos (por ejemplo, del 1000 al 1099). Este método estará basado en dos arrays con los nombres y apellidos, y un generador de números aleatorios, como:

```
class Prueba {
  final static String[] nombres = new String[] { "Ana",
"Elena", "Isabel",
        "Luisa", "María", "Juan", "Carlos", "Miguel",
"Alberto", "José", "Óscar" };
  final static String[] apellidos = new String[] { "López",
"García", "Martínez",
        "González", "Álvarez", "Fernández", "Pérez", "Gómez",
"Rodríguez" };
  final static Random rnd = new Random();
}
```

de forma que los nombres completos de los alumnos puedan ser obtenidos de esta manera:

```
        String nombre = nombres[rnd.nextInt(nombres.length)];
        String apellido =
apellidos[rnd.nextInt(apellidos.length)];
```

Este algoritmo debería evitar los nombres repetidos; antes de insertar un alumno, comprobará que no existe, y generará nuevos nombres para el mismo ID hasta que no esté duplicado.

Pruebas de la clase Gestor

La clase Gestor tendrá varios uno o varios métodos de prueba, que en conjunto cubran la totalidad de las funciones descritas.
Por ejemplo:

```
public class Prueba {
  static void pruebaGestor1() {
        Gestor gestor = new Gestor();
        rellenarAlumnos(gestor, 10001, 100);
```

```
        rellenarAulasYOrdenadores(gestor);
        //
        gestor.ponerAlumnoEnCola(10001);
        gestor.avanzarMinutosTranscurridos (3);
        gestor.ponerAlumnoEnCola (10002);
        gestor.avanzarMinutosTranscurridos(3);
        ...
  }
  public static void main(String[] args) {
        // pruebacolaAlumnos1();
        pruebaGestor1();
        // pruebaGestor2();
        // pruebaGestor3();
  }
}
```

Al probar la clase Gestor, recuerda que sólo se puede llamar a los métodos en dicha clase, esto es por lo que las estructuras de datos fueron declaradas como privadas .

Aclaraciones

Los arrays de Java sólo se pueden usar para:

- Rellenar los árboles de alumnos, como se indicó en el ejemplo anterior.

Las estructuras de datos de Java (ArrayList) sólo podrán ser usadas para:

- Implementar las listas de aulas y de ordenadores.
- Los métodos que devuelvan colecciones (ordenadores libres y ocupados, y alumnos con un apellido dado).

Los árboles de alumnos y la cola de alumnos serán implementados sin usar arrays ni estructuras de datos de Java, sino desarrollando estructuras de datos enlazadas lineales (listas) y no lineales (árboles).

Ejemplo de resultado

Este es un ejemplo de resultado de un método de pruebas de la clase Gestor (es el resultado de la solución, realizada en inglés). El número al comienzo de cada línea representa el número de minutos transcurridos.

```
0: Student 10001-Johnson, John in queue
0: Student 10001-Johnson, John is taking 1.1.A10-01
3: Student 10002-Green, Ann in queue
3: Student 10002-Green, Ann is taking 1.1.A10-02
6: Student 10003-Connery, Ann in queue
6: Student 10003-Connery, Ann is taking 1.1.A10-03
9: Student 10004-Johnson, Charles in queue
9: Student 10004-Johnson, Charles is taking 1.1.A12-01
39: Student 10005-Lee, Sean in queue
39: Student 10005-Lee, Sean is taking 1.1.A14-01
49: Student 10006-Smith, Charles in queue
49: Student 10006-Smith, Charles is taking 1.1.A14-02
79: Student 10007-Connery, Charles in queue
79: Freeing computer 1.1.A10-01; was taken by 10001-Johnson,
John
79: Enqueuing student: 10001-Johnson, John
79: Student 10007-Connery, Charles is taking 1.1.A10-01
   …
169: Enqueuing student: 10008-Robertson, Michael
169: Student 10003-Connery, Ann is taking 1.1.A14-02
169: The student is not in the queue nor working: 10018
172: Student 10019-Edwards, Charles in queue
199: Freeing computer 1.1.A10-01; was taken by 10009-Johnson,
Rachel
199: Enqueuing student: 10009-Johnson, Rachel
199: Student 10005-Lee, Sean is taking 1.1.A10-01
199: Freeing computer 1.1.A10-02; was taken by 10010-Robertson,
Joseph
199: Enqueuing student: 10010-Robertson, Joseph
199: Student 10013-Lee, Charles is taking 1.1.A10-02
199: Freeing computer 1.1.A10-03; was taken by 10011-White,
Sean
199: Enqueuing student: 10011-White, Sean
199: Student 10006-Smith, Charles is taking 1.1.A10-03
```

Solución:

```java
package finalpractice;

public class Student {
	private int id;
	private String lastName;
	private String firstName;

	public int getId() {
		return id;
	}

	public String getLastName() {
		return lastName;
	}

	public String getFirstName() {
		return firstName;
	}

	public Student(int id, String lastName, String firstName) {
		this.id = id;
		this.lastName = lastName;
		this.firstName = firstName;
	}

	public Student(int id) {
		this.id = id;
	}

	public Student(String lastName, String firstName) {
		this.lastName = lastName;
		this.firstName = firstName;
	}

	@Override
	public String toString() {
		return id + "-" + lastName + ", " + firstName;
	}

	@Override
	public boolean equals(Object obj) {
		if (obj instanceof Student) {
			Student student = (Student) obj;
			return this.id == student.id;
		}
		return false;
	}
}
```

//

```java
package finalpractice;

public class Room {
	String name;
	boolean closed = false;
```

```
	ComputerList computerList = new ComputerList();

	public Room(String name) {
		this.name = name;
	}

	public String toString() {
		return name;
	}

	public Computer addComputer(int id) {
		Computer ordenador = new Computer(this.name, id);
		this.computerList.add(ordenador);
		return ordenador;
	}

	public void open() {
		this.closed = false;
	}

	public void close() {
		this.closed = true;
	}

	public boolean isClosed() {
		return this.closed;
	}
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package finalpractice;

import java.text.DecimalFormat;

public class Computer {
	String roomName;
	int id;

	Student student;
	int minutesPassed; // minutes passed when the computer was
taken

	public Computer(String roomName, int id) {
		this.roomName = roomName;
		this.id = id;
	}

	public String toString() {
		return roomName + "-" + new
DecimalFormat("00").format(id);
	}

	public void take(Student student, int minutesPassed) {
		assert (this.student == null);
		this.student = student;
		this.minutesPassed = minutesPassed;
	}
```

```
	public void leave() {
		assert (this.student != null);
		this.student = null;
	}

	public boolean isTaken() {
		return this.student != null;
	}

}
////////////////////////////////////////////////////////////////////////////////////////////////////

package finalpractice;

import java.util.ArrayList;
import java.util.List;

import tree.generic.BsTree;
import tree.generic.BsNode;

public class StudentTreeByName extends BsTree<Student> {
	/*
	 * compare by name
	 */
	@Override
	protected int compare(Student student1, Student student2) {
		int r = 0;
		if (r == 0) {
			r = student1.getLastName().compareToIgnoreCase(
					student2.getLastName());
		}
		if (r == 0) {
			r = student1.getFirstName().compareToIgnoreCase(
					student2.getFirstName());
		}
		return r;
	}

	/*
	 * get student by name
	 */

	public Student getStudent(String lastName, String firstName)
{
		Student student = new Student(lastName, firstName);
		return this.search(student);
	}

	/*
	 * Get students with a given last name
	 */

	public List<Student> getStudentListByLastName(String
lastName) {
		// N
		List<Student> studentList = new ArrayList<Student>();
		getStudentListByLastNameRec(this.getRoot(), lastName,
studentList);
```

```
			return studentList;
		}

		private void getStudentListByLastNameRec(BsNode<Student>
node,
				String lastName, List<Student> studentList) {
			if (node != null) {
				Student student = node.getObj();
				// in-order -> alphabetical order
				getStudentListByLastNameRec(node.getLeftNode(),
lastName,
						studentList);
				if
(lastName.equalsIgnoreCase(student.getLastName())) {
					studentList.add(student);
				}
				getStudentListByLastNameRec(node.getRightNode(),
lastName,
						studentList);
			}
		}

}
```

//

```
package finalpractice;

import tree.generic.AvlTree;

public class StudentTreeById extends AvlTree<Student> {

		/*
		 * compare by id
		 */
		@Override
		protected int compare(Student student1, Student student2) {
			return student1.getId() - student2.getId();
		}

		/*
		 * get student by id
		 */

		public Student getStudent(int id) {
			Student student=new Student(id);
			return this.search(student);
		}


}
```

//

```
package finalpractice;

import java.util.ArrayList;

@SuppressWarnings("serial")
```

```java
public class RoomList extends ArrayList<Room> {
	public Room getByName(String name) {
		for (Room room : this) {
			if (room.name.equalsIgnoreCase(name)) {
				return room;
			}
		}
		return null;
	}

}
```

//

```java
package finalpractice;

import java.util.ArrayList;

@SuppressWarnings("serial")
public class ComputerList extends ArrayList<Computer> {

}
```

//

```java
package finalpractice;

import list.generic.ColaExt;

public class StudentQueue extends ColaExt<Student> {
	public Student peekFirstStudent() throws Exception {
		return this.front();
	}

	public Student dequeueFirstStudent() throws Exception {
		return this.dequeue();
	}

	public void enqueueStudent(Student student) throws Exception
{
		this.enqueue(student);
	}

	public Student removeStudentById(int id) throws Exception {
		Student alumno = new Student(id);
		return this.remove(alumno);
	}
}
```

//

```java
package finalpractice;

import java.util.ArrayList;
import java.util.List;

public class Manager {

	private StudentTreeByName studentTreeByName = new
StudentTreeByName();
```

```
	private StudentTreeById studentTreeById = new
StudentTreeById();

	private RoomList roomList = new RoomList();

	private StudentQueue studentQueue = new StudentQueue();

	private int minutesPassed;

	final int maxMinutes = 60;

	/*
	 * Adds students
	 */

	public boolean canAddStudent(Student student) {
		// average log(N), worst N
		boolean can = false;
		can |=
this.studentTreeByName.getStudent(student.getLastName(),
				student.getFirstName()) != null;
		can |=
this.studentTreeById.getStudent(student.getId()) != null;
		return !can;
	}

	public void addStudent(Student student) throws Exception {
		// average log(N), worst N
		this.studentTreeByName.add(student);
		// log(N)
		this.studentTreeById.add(student);
	}

	/*
	 * Adds rooms and computers
	 */

	public void addRoom(String roomName, int numComputers) {
		Room room = new Room(roomName);
		for (int id = 1; id < numComputers; ++id) {
			Computer computer = new Computer(roomName, id);
			room.computerList.add(computer);
		}
		this.roomList.add(room);
	}

	public Room getRoomByName(String name) {
		Room room = this.roomList.getByName(name);
		return room;
	}

	/*
	 * Basic functionality
	 */

	public Student getStudentById(int id) {
		// log(N)
		Student student = new Student(id);
		return this.studentTreeById.search(student);
```

```
	}

	public Student getStudentByName(String lastName, String
firstName) {
		// average log(N), worst N
		Student student = new Student(lastName, firstName);
		return this.studentTreeByName.search(student);
	}

	public List<Student> getStudentListByLastName(String
lastName) {
		// N
		List<Student> listaAlumnos = this.studentTreeByName
				.getStudentListByLastName(lastName);
		return listaAlumnos;
	}

	public Student putStudentInQueue(int id) {
		// log(N), being N = n. students
		Student student = getStudentById(id);
		enqueueStudent(student);
		return student;
	}

	public void enqueueStudent(Student student) {
		// 1
		if (student == null) {
			System.out
					.println(this.minutesPassed + ": " +
"Can't find student");
		} else {
			System.out.println(this.minutesPassed + ": " +
"Student " + student
					+ " in queue");
			this.studentQueue.enqueue(student);
			manageQueue();
		}
	}

	public Student getFirstStudentInQueue() {
		// 1
		try {
			return this.studentQueue.front();
		} catch (Exception e) {
			System.out.println(this.minutesPassed + ": "
					+ "Student queue is empty");
			return null;
		}
	}

	public Student removeFirstStudentFromQueue() {
		// 1
		try {
			return this.studentQueue.dequeueFirstStudent();
		} catch (Exception e) {
			System.out.println(this.minutesPassed + ": "
					+ "Student queue is empty");
			return null;
		}
```

```
	}

	public Student removeStudentFromQueueOrComputer(int id) {
		Student student = null;
		try {
			student =
this.studentQueue.removeStudentById(id);
			System.out.println(this.minutesPassed + ": "
					+ "Student removed from queue: " +
student);
			if (student != null) {
				return student;
			}
		} catch (Exception e) {
		}
		for (Room room : roomList) {
			for (Computer computer : room.computerList) {
				if (computer.student != null) {
					if (computer.student.getId() == id)
{
						student =
freeComputer(computer);

	System.out.println(this.minutesPassed + ": "
							+ "Student " +
student + " is leaving "
							+ computer);
						return student;
					}
				}
			}
		}
		System.out.println(this.minutesPassed + ": "
				+ "The student is not in the queue nor
working: " + id);
		return null;
	}

	public List<Computer> getFreeComputerList() {
		// N, N = n. computers
		List<Computer> result = new ArrayList<Computer>();
		for (Room room : roomList) {
			if (!room.isClosed()) {
				for (Computer computer :
room.computerList) {
					if (!computer.isTaken()) {
						result.add(computer);
					}
				}
			}
		}
		return result;
	}

	public List<Computer> getTakenComputerList() {
		// N, N = n. computers
		List<Computer> result = new ArrayList<Computer>();
		for (Room room : roomList) {
			for (Computer computer : room.computerList) {
```

```
                        if (computer.isTaken()) {
                              result.add(computer);
                        }
                  }
            }
            return result;
      }

      public Computer getComputerTakenForMostTime() {
            // N, N = n. computers
            List<Computer> list = getTakenComputerList();
            Computer result = null;
            for (Computer computer : list) {
                  if (result == null) {
                        result = computer;
                  } else {
                        if (computer.minutesPassed <
result.minutesPassed) {
                              result = computer;
                        }
                  }
            }
            return result;
      }

      public List<Student> getWorkingStudentList() {
            // N, N = n. computers
            List<Student> result = new ArrayList<Student>();
            for (Room room : roomList) {
                  for (Computer computer : room.computerList) {
                        if (computer.isTaken()) {
                              result.add(computer.student);
                        }
                  }
            }
            return result;
      }

      /*
       * Queue management
       */

      public void manageQueue() {
            // while there are students in the queue
            while (!studentQueue.isEmpty()) {
                  // look for a free computer
                  Computer freeComputer = null;
                  List<Computer> freeComputerList =
getFreeComputerList();
                  if (!freeComputerList.isEmpty()) {
                        freeComputer = freeComputerList.get(0);
                  } else {
                        // if there is not a free computer, look
for a freeable one
                        Computer takenComputer =
getComputerTakenForMostTime();
                        if (takenComputer != null) {
                              // if there is a freeable computer
and the student has
```

```
                              // exceeded use time
                              if (this.minutesPassed -
takenComputer.minutesPassed >= maxMinutes) {
                                    // free the computer
                                    Student student =
takenComputer.student;

      System.out.println(this.minutesPassed + ": "
                                                + "Freeing
computer " + takenComputer
                                                + "; was taken by
" + student);
                                    takenComputer.leave();
                                    freeComputer = takenComputer;

      System.out.println(this.minutesPassed + ": "
                                                + "Enqueuing
student: " + student);

      this.studentQueue.enqueue(student);
                                    }
                              }
                        }
                        // if there is no free/freeable computer, return
                        // else, let the first student in the queue take
it
                        if (freeComputer == null) {
                              break;
                        } else {
                              try {
                                    Student student =
studentQueue.dequeueFirstStudent();
                                    takeComputer(freeComputer, student);
                              } catch (Exception e) {
                                    e.printStackTrace(); // this
shouldn't happen
                              }

                        }
                  }
      }

      public void openRoom(String roomName) {
            Room room = this.roomList.getByName(roomName);
            System.out.println(this.minutesPassed + ": " +
"Opening room: " + room);
            room.open();
            manageQueue();
      }

      public void closeRoom(String roomName) {
            Room room = this.roomList.getByName(roomName);
            System.out.println(this.minutesPassed + ": " +
"Closing room: " + room);
            for (Computer computer : room.computerList) {
                  if (computer.isTaken()) {
                        Student student = computer.student;
                        computer.leave();
```

```
                        System.out.println(this.minutesPassed + ":
"
                                    + "Putting student in queue: "
+ student);
                        this.studentQueue.enqueue(student);
                  }
            }
            room.close();
            manageQueue();
      }

      private void takeComputer(Computer computer, Student
student) {
            System.out.println(this.minutesPassed + ": " +
"Student " + student
                        + " is taking " + computer);
            computer.take(student, this.minutesPassed);
      }

      private Student freeComputer(Computer computer) {
            Student student = computer.student;
            computer.leave();
            return student;
      }

      /*
       * Time managemnent
       */

      public void advanceMinutesPassed(int minutes) {
            while (minutes > 0) {
                  ++this.minutesPassed;
                  --minutes;
                  manageQueue();
            }
      }

}
////////////////////////////////////////////////////////////////////////////////////////////////////

package finalpractice;

import java.util.Random;

import tree.generic.AvlTree;
import tree.generic.BsTree;

public class Tester {

      static void testAvlTree() {
            try {
                  AvlTree<Integer> tree = new AvlTree<Integer>();
                  tree.add(0);
                  tree.show();
                  tree.add(1);
                  tree.show();
                  tree.add(2);
                  tree.show();
```

```
                tree.add(3);
                tree.show();
                tree.add(4);
                tree.show();
                tree.add(5);
                tree.show();
                tree.add(6);
                tree.show();
                tree.add(7);
                tree.show();
                tree.add(8);
                tree.show();
                tree.add(9);
                tree.show();
          } catch (Exception e) {
                e.printStackTrace();
          }
    }

    static void testBsTree() {
          try {
                BsTree<Integer> tree = new BsTree<Integer>();
                tree.add(3);
                System.out.println(tree.toString());
                tree.add(2);
                System.out.println(tree.toString());
                tree.add(6);
                System.out.println(tree.toString());
                tree.add(1);
                System.out.println(tree.toString());
                tree.add(4);
                System.out.println(tree.toString());
                tree.add(9);
                System.out.println(tree.toString());
                tree.add(8);
                System.out.println(tree.toString());
                tree.add(7);
                System.out.println(tree.toString());
                tree.add(5);
                System.out.println(tree.toString());
                tree.add(0);
                System.out.println(tree.toString());
          } catch (Exception e) {
                e.printStackTrace();
          }
    }

    /*
     * Random student generator
     */

    static final String[] firstNames = new String[] { "Ann",
"Helen",
                "Elizabeth", "Mary", "Rachel", "John",
"Charles", "Michael",
                "Albert", "Joseph", "Sean" };
    static final String[] lastNames = new String[] {
"Robertson", "Smith",
```

```
                  "Lee", "White", "Owens", "Black", "Brown",
"Edwards", "Connery",
                  "Johnson", "Green" };
      static final Random rnd = new Random();

      static void populateStudents(Manager manager, int id1, int
n) {
            // average N log(N), worst N
            for (int i = 0; i < n; ++i) {
                  Student student;
                  do {
                        int id = id1;
                        String firstName =
firstNames[rnd.nextInt(firstNames.length)];
                        String lastName =
lastNames[rnd.nextInt(lastNames.length)];
                        student = new Student(id, lastName,
firstName);
                  } while (!manager.canAddStudent(student));
                  //
                  try {
                        manager.addStudent(student);
                  } catch (Exception e) {
                        e.printStackTrace(); // this shouldn't
happen
                  }
                  //
                  id1++;
            }
      }

      static void populateRoomsAndComputers(Manager manager) {
            manager.addRoom("1.1.A10",4);
            manager.addRoom("1.1.A12",2);
            manager.addRoom("1.1.A14",3);
      }

      static void testManager1() {
            Manager manager = new Manager();
            //
            populateStudents(manager, 10001, 100);
            populateRoomsAndComputers(manager);
            //
            manager.putStudentInQueue(10001);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10002);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10003);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10004);
            manager.advanceMinutesPassed(30);
            manager.putStudentInQueue(10005);
            manager.advanceMinutesPassed(10);
            manager.putStudentInQueue(10006);
            manager.advanceMinutesPassed(30);
            manager.putStudentInQueue(10007);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10008);
            manager.advanceMinutesPassed(3);
```

```
            manager.putStudentInQueue(10009);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10010);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10011);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10012);
            manager.advanceMinutesPassed(3);
            manager.closeRoom("1.1.A12");
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10013);
            manager.advanceMinutesPassed(30);
            manager.putStudentInQueue(10014);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10015);
            manager.advanceMinutesPassed(3);
            manager.removeStudentFromQueueOrComputer(10010);
            manager.advanceMinutesPassed(30);
            manager.putStudentInQueue(10017);
            manager.advanceMinutesPassed(3);
            manager.removeStudentFromQueueOrComputer(10018);
            manager.advanceMinutesPassed(3);
            manager.putStudentInQueue(10019);
            manager.advanceMinutesPassed(30);
      }

      public static void main(String[] args) {
            // testBsTree();
            // testAvlTree();
            testManager1();
      }

}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package tree.generic;

public class AvlNode<T> extends BsNode<T> {
      public AvlNode(T obj) {
            super(obj);
      }

      public AvlNode<T> getLeftNode() {
            return (AvlNode<T>)this.leftNode;
      }

      public AvlNode<T> getRightNode() {
            return (AvlNode<T>)this.rightNode;
      }

      /*
       * Height management
       */

      int height;

      void updateHeight() {
            this.height = 0;
```

```
            if (this.leftNode != null) {
                  this.height = Math.max(this.height, 1 +
getLeftNode().height);
            }
            if (this.rightNode != null) {
                  this.height = Math.max(this.height, 1 +
getRightNode().height);
            }
      }
}
////////////////////////////////////////////////////////////////////////////////////////////////////

package tree.generic;

//public class ArbolAVL<T extends Comparable<T>> {
public class AvlTree<T> extends BsTree<T> {
      public AvlNode<T> getRoot() {
            return (AvlNode<T>)this.root;
      }

      /*
       * Recursive AVL insertion
       */

      @Override
      public void add(T obj) throws Exception {
            this.root = addRec(getRoot(), obj);
      }

      private AvlNode<T> addRec(AvlNode<T> node, T obj) throws
Exception {
            if (node == null) {
                  AvlNode<T> newNode = new AvlNode<T>(obj);
                  return newNode;
            } else {
                  int cmp = compare(obj, node.obj);
                  if (cmp > 0) {
                        node.rightNode =
addRec(node.getRightNode(), obj);
                        node.updateHeight();
                        node = balanceRight(node);
                  } else if (cmp < 0) {
                        node.leftNode = addRec(node.getLeftNode(),
obj);
                        node.updateHeight();
                        node = balanceLeft(node);
                  } else {
                        throw new Exception("obj lready exists");
                  }
                  return node;
            }
      }

      private AvlNode<T> balanceLeft(AvlNode<T> node) {
            AvlNode<T> leftNode = (AvlNode<T>)node.leftNode;
            AvlNode<T> rightNode= (AvlNode<T>)node.rightNode;
            int leftHeight = (leftNode == null) ? 0 :
leftNode.height;
```

```
            int rightHeight = (rightNode == null) ? 0 :
rightNode.height;
            if (leftHeight > rightHeight + 1) {
                  AvlNode<T> leftLeftNode =
(AvlNode<T>)leftNode.leftNode;
                  AvlNode<T> leftRightNode=
(AvlNode<T>)leftNode.rightNode;
                  int leftLeftHeight = (leftLeftNode == null) ? 0
                              : leftLeftNode.height;
                  int leftRightHeight = (leftRightNode == null) ?
0
                              : leftRightNode.height;
                  if (leftLeftHeight > leftRightHeight) {
                        node = balanceLeftLeft(node);
                  } else {
                        node = balanceLeftRight(node);
                  }
            }
            return node;
      }

      private AvlNode<T> balanceRight(AvlNode<T> node) {
            AvlNode<T> leftNode = (AvlNode<T>)node.leftNode;
            AvlNode<T> rightNode= (AvlNode<T>)node.rightNode;
            int leftHeight = (leftNode == null) ? 0 :
leftNode.height;
            int rightHeight = (rightNode == null) ? 0 :
rightNode.height;
            if (rightHeight > leftHeight + 1) {
                  AvlNode<T> rightLeftNode =
(AvlNode<T>)rightNode.leftNode;
                  AvlNode<T> rightRightRight=
(AvlNode<T>)rightNode.rightNode;
                  int rightLeftHeight = (rightLeftNode == null) ?
0
                              : rightLeftNode.height;
                  int rightRightHeight = (rightRightRight == null)
? 0
                              : rightRightRight.height;
                  if (rightLeftHeight > rightRightHeight) {
                        node = balanceRightLeft(node);
                  } else {
                        node = balanceRightRight(node);
                  }
            }
            return node;
      }

      private AvlNode<T> balanceLeftLeft(AvlNode<T> node) {
            AvlNode<T> node6 = node;
            AvlNode<T> node4 = (AvlNode<T>)node6.leftNode;
            AvlNode<T> node2 = (AvlNode<T>)node4.leftNode;
            AvlNode<T> node7 = (AvlNode<T>)node6.rightNode;
            AvlNode<T> node5 = (AvlNode<T>)node4.rightNode;
            AvlNode<T> node3 = (AvlNode<T>)node2.rightNode;
            AvlNode<T> node1 = (AvlNode<T>)node2.leftNode;
            node2.leftNode = node1;
            node2.rightNode = node3;
            node2.updateHeight();
```

```
        node6.leftNode = node5;
        node6.rightNode = node7;
        node6.updateHeight();
        node4.leftNode = node2;
        node4.rightNode = node6;
        node4.updateHeight();
        return node4;
    }

    private AvlNode<T> balanceLeftRight(AvlNode<T> node) {
        AvlNode<T> node6 = node;
        AvlNode<T> node2 = (AvlNode<T>)node6.leftNode;
        AvlNode<T> node4 = (AvlNode<T>)node2.rightNode;
        AvlNode<T> node7 = (AvlNode<T>)node6.rightNode;
        AvlNode<T> node5 = (AvlNode<T>)node4.rightNode;
        AvlNode<T> node3 = (AvlNode<T>)node4.leftNode;
        AvlNode<T> node1 = (AvlNode<T>)node2.leftNode;
        node2.leftNode = node1;
        node2.rightNode = node3;
        node2.updateHeight();
        node6.leftNode = node5;
        node6.rightNode = node7;
        node6.updateHeight();
        node4.leftNode = node2;
        node4.rightNode = node6;
        node4.updateHeight();
        return node4;
    }

    private AvlNode<T> balanceRightLeft(AvlNode<T> node) {
        AvlNode<T> node2 = node;
        AvlNode<T> node6 = (AvlNode<T>)node2.rightNode;
        AvlNode<T> node4 = (AvlNode<T>)node6.leftNode;
        AvlNode<T> node1 = (AvlNode<T>)node2.leftNode;
        AvlNode<T> node3 = (AvlNode<T>)node4.leftNode;
        AvlNode<T> node5 = (AvlNode<T>)node4.rightNode;
        AvlNode<T> node7 = (AvlNode<T>)node6.rightNode;
        node2.leftNode = node1;
        node2.rightNode = node3;
        node2.updateHeight();
        node6.leftNode = node5;
        node6.rightNode = node7;
        node6.updateHeight();
        node4.leftNode = node2;
        node4.rightNode = node6;
        node4.updateHeight();
        return node4;
    }

    private AvlNode<T> balanceRightRight(AvlNode<T> node) {
        AvlNode<T> node2 = node;
        AvlNode<T> node4 = (AvlNode<T>)node2.rightNode;
        AvlNode<T> node6 = (AvlNode<T>)node4.rightNode;
        AvlNode<T> node1 = (AvlNode<T>)node2.leftNode;
        AvlNode<T> node3 = (AvlNode<T>)node4.leftNode;
        AvlNode<T> node5 = (AvlNode<T>)node6.leftNode;
        AvlNode<T> node7 = (AvlNode<T>)node6.rightNode;
        node2.leftNode = node1;
        node2.rightNode = node3;
```

```
            node2.updateHeight();
            node6.leftNode = node5;
            node6.rightNode = node7;
            node6.updateHeight();
            node4.leftNode = node2;
            node4.rightNode = node6;
            node4.updateHeight();
            return node4;
      }

}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package tree.generic;

public class BsNode<T> {
      T obj;
      BsNode<T> leftNode;
      BsNode<T> rightNode;

      public T getObj() {
            return this.obj;
      }

      public BsNode<T> getLeftNode() {
            return this.leftNode;
      }

      public BsNode<T> getRightNode() {
            return this.rightNode;
      }

      public BsNode(T obj) {
            this.obj = obj;
      }

      public String toString() {
            return "[" + this.obj.toString() + "]";
      }

}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package tree.generic;

// public class Arbol<T extends Comparable<T>> {
public class BsTree<T> {
      BsNode<T> root;

      public BsNode<T> getRoot() {
            return this.root;
      }

      /*
       * Recursive insertion
       */

      public void add(T obj) throws Exception {
```

```
            this.root = addRec(this.root, obj);
      }

      private BsNode<T> addRec(BsNode<T> node, T obj) throws
Exception {
            if (node == null) {
                  BsNode<T> nodoNuevo = new BsNode<T>(obj);
                  return nodoNuevo;
            } else {
                  int cmp = compare(obj, node.obj);
                  if (cmp > 0) {
                        node.rightNode = addRec(node.rightNode,
obj);
                  } else if (cmp < 0) {
                        node.leftNode = addRec(node.leftNode,
obj);
                  } else {
                        throw new Exception("obj already exists");
                  }
                  return node;
            }
      }

      /*
       * Non-recursive insertion (not used)
       */
      void añadirNoRec(T obj) throws Exception {
            BsNode<T> newNode = new BsNode<T>(obj);
            BsNode<T> node = root;
            if (node == null) {
                  this.root = newNode;
            } else {
                  while (true) {
                        int cmp = compare(obj, node.obj);
                        if (cmp > 0) {
                              if (node.rightNode == null) {
                                    node.rightNode = newNode;
                                    break;
                              } else {
                                    node = node.rightNode;
                              }
                        } else if (cmp < 0) {
                              if (node.leftNode == null) {
                                    node.leftNode = newNode;
                                    break;
                              } else {
                                    node = node.leftNode;
                              }
                        } else {
                              throw new Exception("obj already
exists");
                        }
                  }
            }
      }

      /*
       * Recursive search
       */
```

```java
public boolean isObjInTree(T obj) {
	return searchRec(this.root, obj) != null;
}

public T search(T obj) {
	return searchRec(this.root, obj);
}

private T searchRec(BsNode<T> node, T obj) {
	if (node == null) {
		return null;
	} else {
		int cmp = compare(obj, node.obj);
		if (cmp > 0) {
			return searchRec(node.rightNode, obj);
		} else if (cmp < 0) {
			return searchRec(node.leftNode, obj);
		} else {
			return node.obj;
		}
	}
}

/*
 * Element comparer. Overridable (@Override). By default,
 * it uses Comparable<T>.compare
 */

@SuppressWarnings("unchecked")
protected int compare(T obj1, T obj2) {
	if (obj1 instanceof Comparable<?>) {
		Comparable<T> obj1c = (Comparable<T>) obj1;
		return obj1c.compareTo(obj2);
	}
	return 0;
}

/*
 * For debugging purposes
 */

public void show() {
	System.out.println(this.toString());
}

@Override
public String toString() {
	return toStringRec(this.root);
}

public String toStringRec(BsNode<T> node) {
	if (node == null) {
		return "#";
	} else {
		// pre-order
		String str = "[" + node.obj + "]";
		str += toStringRec(node.leftNode);
		str += toStringRec(node.rightNode);
```

```
                return str;
            }
        }
    }
```