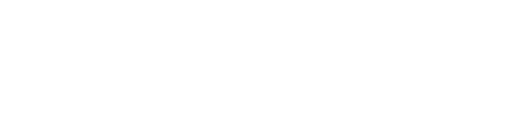
MEMORIA DE LA PRACTICA

INDIVIDUAL 2

José Manuel Pérez Álvarez



# Índice

Creación del árbol……………………………………………2

Ejercicio 3……………………………………………………....3

Complejidad Ejercicio 3………………….……………………..4

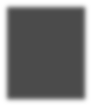
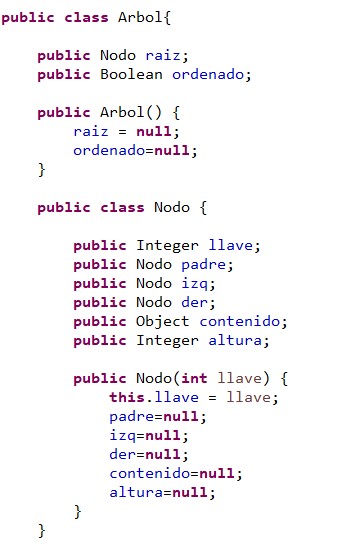
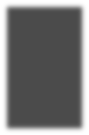
Volcado en pantalla ejercicio 3…....……………………………4

Ejercicio 4……………………………………………………….5

Complejidad Ejercicio 4……………………………………….....6

Volcado en pantalla ejercicio 4…………………………………..6

## CREACIÓN DE ÁRBOL BINARIO



Empezaremos por crear una clase

Árbol en la que tendrá dos

atributos, uno es la raíz del árbol y el otro para ver si el árbol esta ordenado o no, según las indicaciones del ejercicio. Además de un constructor que inicia dichos atributos.

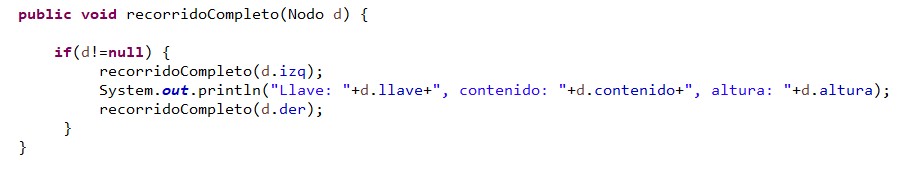
Continuamos con la creación de la clase Nodo que contiene: la llave, el Nodo padre, Nodo hijo izquierdo y derecho, un contenido y la altura en el árbol. Y por último un constructor para inicial los atributos.

Continuaremos con un método insertar, en el que añade un nodo al árbol.

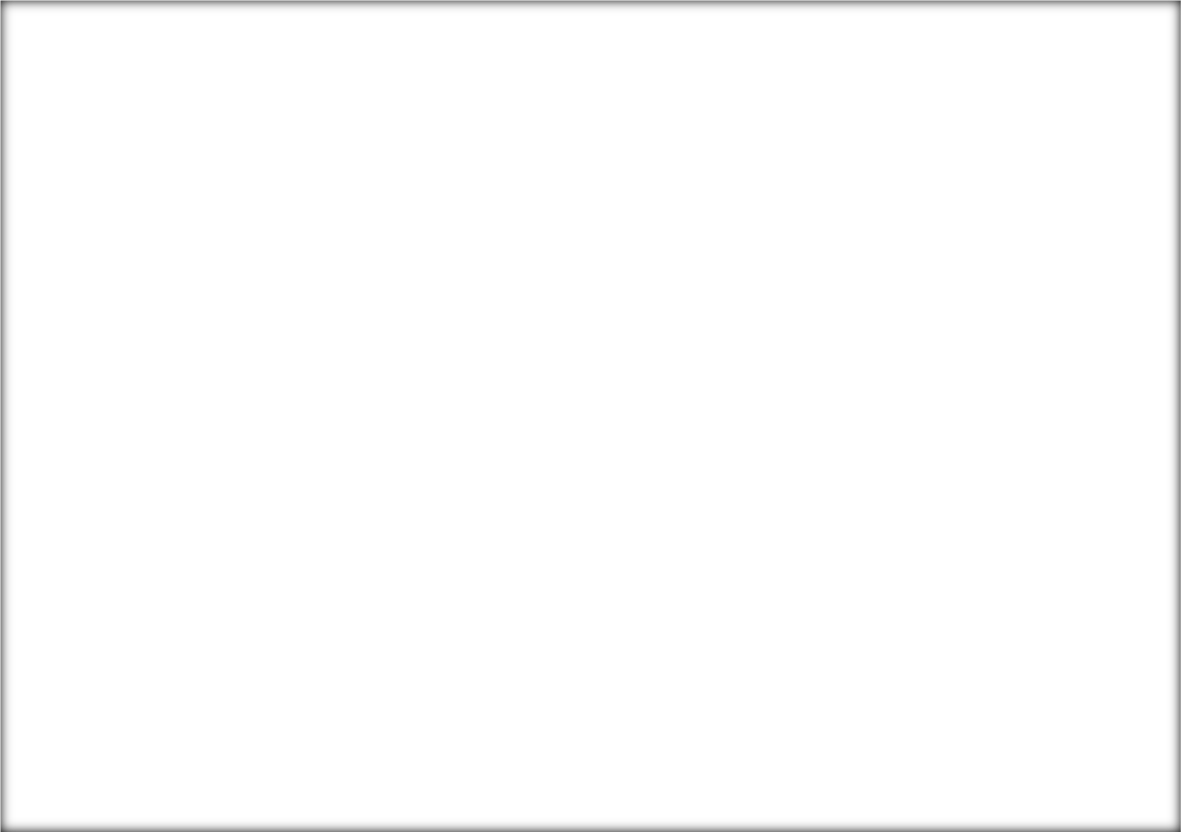
En este método lo añadiremos de manera que vaya buscando el lugar donde insertarlo mediante un nodo auxiliar, dependiendo del valor de las llaves irá hacia el nodo hijo derecho o izquierdo.

Una vez encontrado su lugar se inserta el nodo.

Además, utilizaremos este método para recorrer y mostrar el árbol pasando por parámetro el nodo raíz del árbol.



## EJERCICIO 3



El ejercicio 3 nos indica las condiciones que han de cumplirse para que un árbol esté ordenado, para ello usaremos el método mostrado.

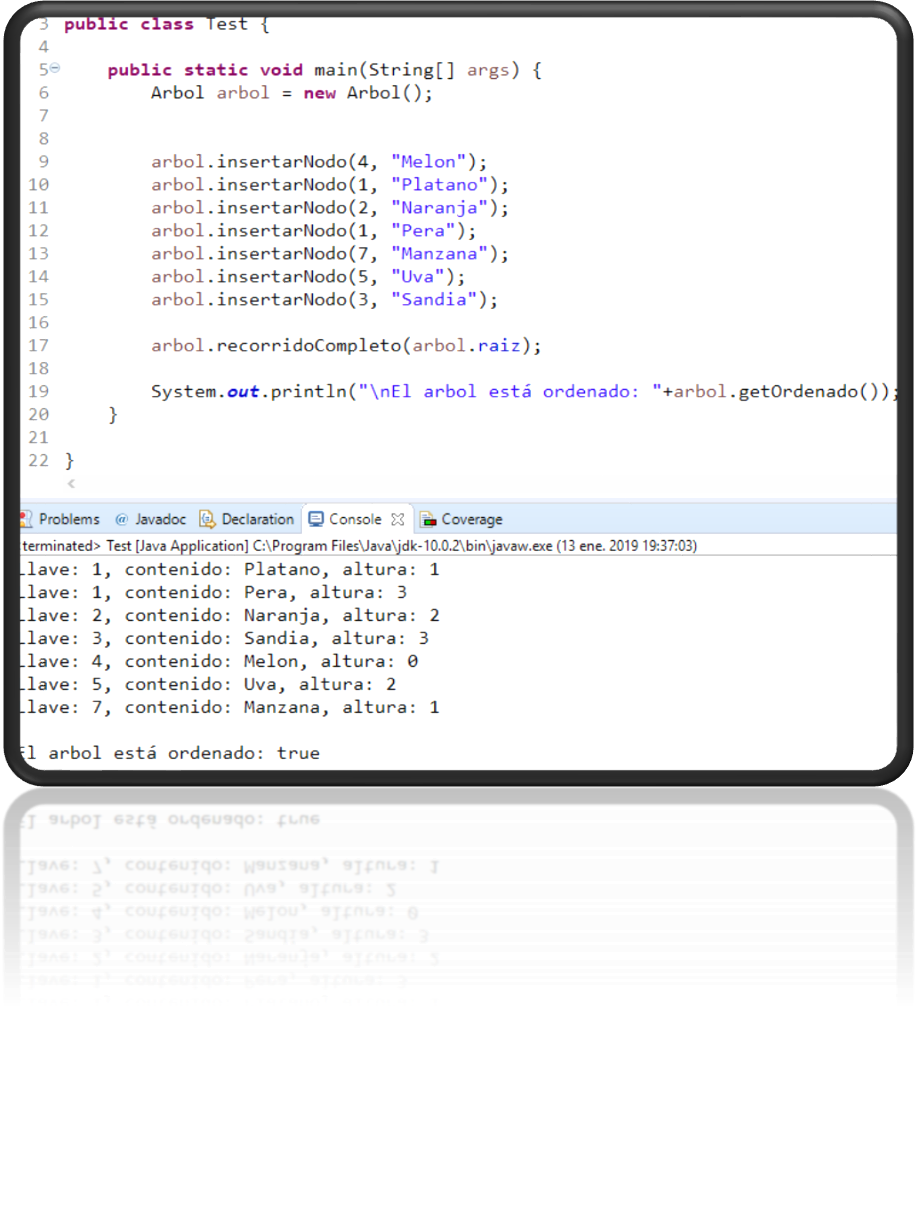
Si el nodo raíz es null quiere decir que el árbol está vacío y por tanto ordenado, y si tiene raíz, iniciaremos como ordenado hasta que vaya recorriendo los demás nodos y se demuestre lo contrario.

Deberemos comprobar que su raíz es mayor o igual a todas las etiquetas de su hijo izquierdo y menor o igual a todas las etiquetas de su hijo derecho. Y también sus hijos deben estar ordenados.

Por último usaremos un getOrdenado() para ejecutar el método y que nos devuelva el valor de dicho atributo.

## Complejidad Ejercicio 3

Para esta complejidad sabemos que el recorrer el árbol tiene una complejidad de ø(n). Tendremos una complejidad:

 𝑇(𝑛) = 𝑇 (𝑛) + 𝑇 (𝑛) ≌ 2𝑇 (𝑛) 𝜖 ø(𝑛)

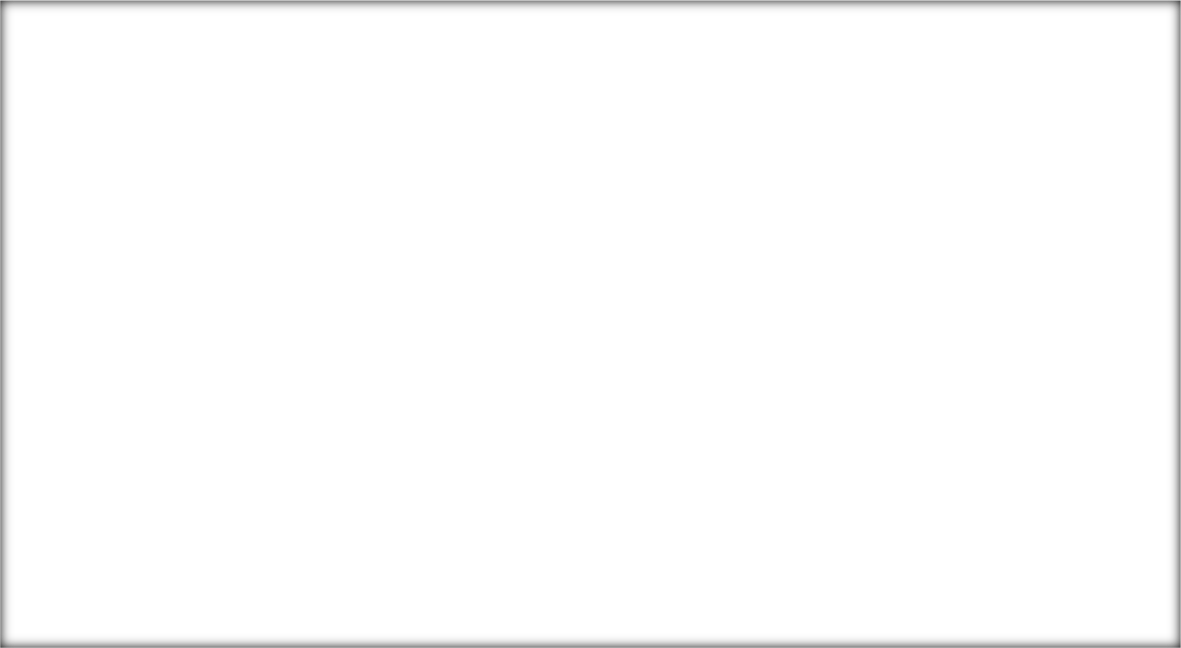
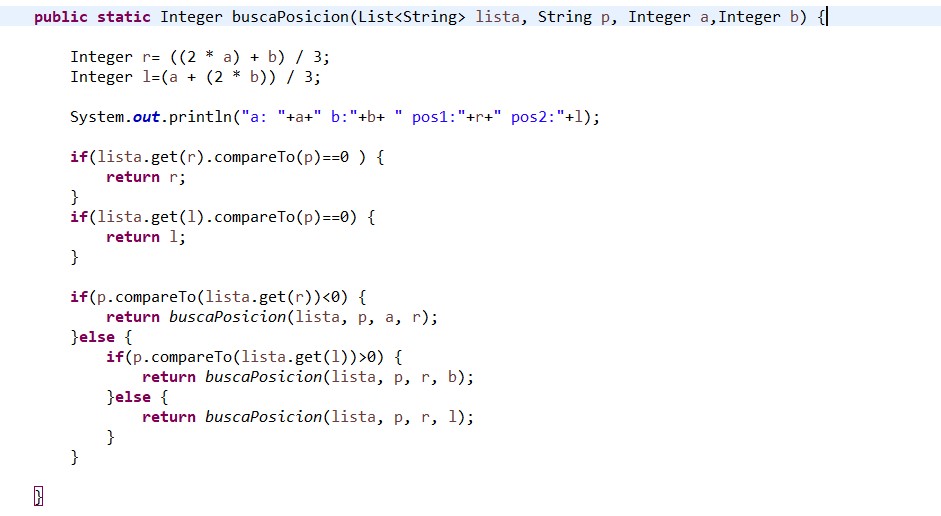
2 2 2

Donde a=2, b=2, d=1, p=0

En cuanto al caso mejor, será cuando el árbol esté vacío. El caso peor dependerá de la altura, cuanta mayor altura mayor complejidad.

## VOLCADO EN PANTALLA

EJERCICIO 4

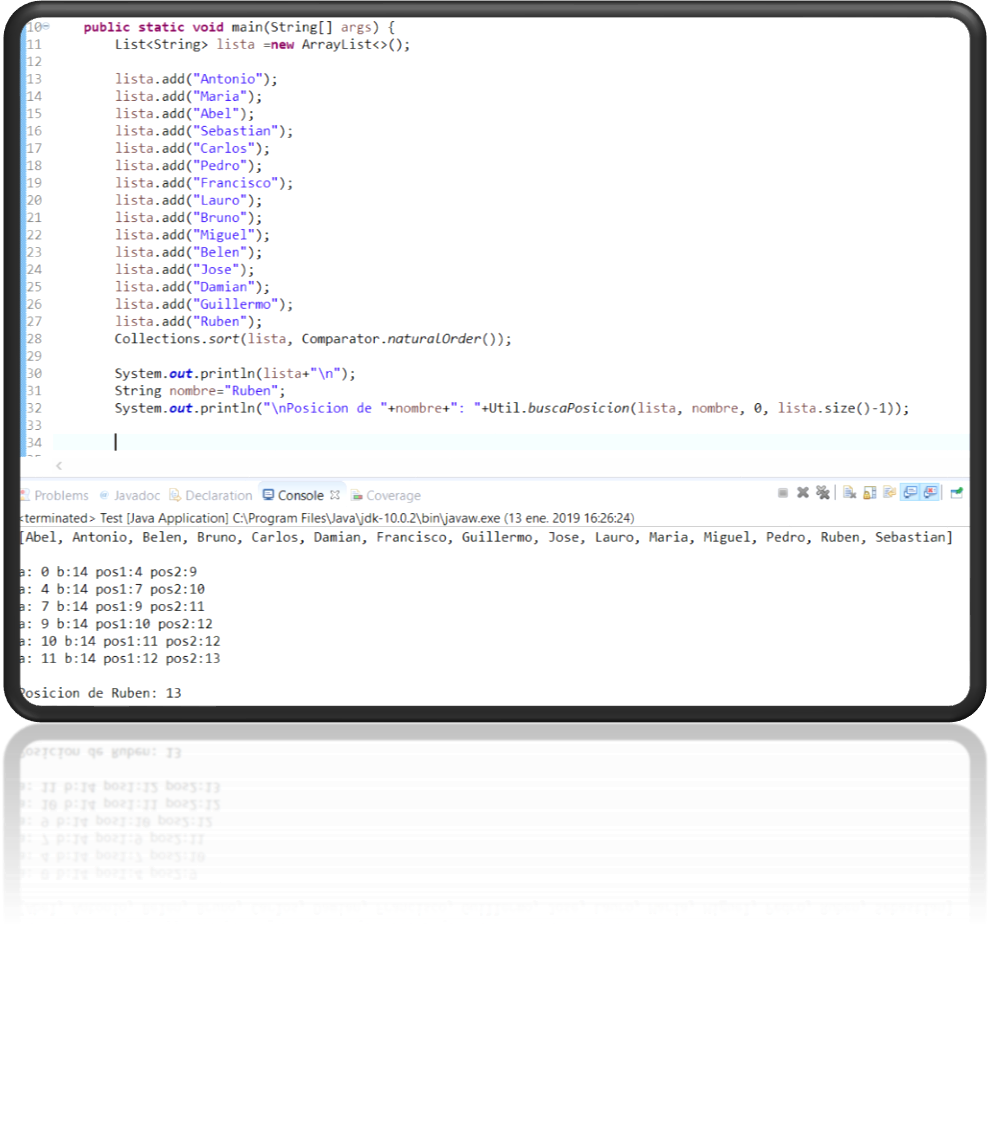


Dada la palabra ‘p’ pasada por parámetros, procederemos a buscar en qué posición de la lista se encuentra, dividiendo la tabla en tres y buscar en que sección se encuentra nuestra palabra, para ello usaremos el parámetro ‘a’ que será la primera posición de la tabla y ‘b’ que será la última.

Lo primero será tomar las variables ‘r’ y ‘l’ como la posición del primer tercio de la lista y el segundo tercio respectivamente. Como la lista está ordenada alfabéticamente haremos uso del compareTo para saber si la palabra esta en las posiciones r o l, y sino buscar entre las distintas divisiones: a-r, r-l, l-b. En el caso de que no pertenezca a la lista se devolverá -1.

## Complejidad Ejercicio 4

La complejidad dependerá de cuantas divisiones de la lista tengamos, además de los valores de máximo y mínimo que van variando por interacciones. Sabiendo que al comparar estos valores la complejidad será ø(logn). Por lo que:

 𝑇(𝑛) = log 𝑛 + 𝑇 (𝑛) + 𝑇 (𝑛) + 𝑇 (𝑛) ≌ log 𝑛 + 3𝑇 (𝑛) 𝜖 ø(n)

3 3 3 3

Donde a=3, b=3, d=0 y p=1

El caso mejor es cuando la palabra no se encuentra en la lista y se devuelve -1. El caso peor depende del tamaño de la tabla, a mayor tabla mayor tiempo de búsqueda.

## VOLCADO EN PANTALLA