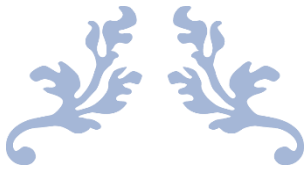


TECNICAS DE PROGRAMACION AVANZADAS

PRACTICA INDIVIDUAL 2

VICTOR PEREZ PEREZ

13/06/2021

Nº exp: 21923658

Contenido

EJERCICIO 1

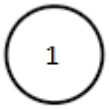
Genera el árbol AVL en el que, partiendo de un árbol vacío, se insertan (en el orden indicado) los siguientes elementos: {1,2,3,4,5,6,7,15,14,13,12,11,10,9,8}.Indica cuál es el árbol AVL final y cuáles han sido las rotaciones resultantes que has necesitado aplicar.

Para empezar el ejercicio, insertaremos los valores uno a uno, e iremos calculando en cada momento el factor de equilibrio (Fe = (Nodos hijo izquierdo – Nodos hijo derecho)) de cada uno de los nodos para saber si tenemos que realizar alguna rotación.

Posibles valores del FE:

- -1: árbol cargado a la derecha. No hace falta rotar.
- 1: árbol cargado a la izquierda. No hace falta rotar.
- 0: árbol balanceado correctamente. No hace falta rotar.
- Si obtenemos otro valor, necesitaremos realizar alguna rotación.

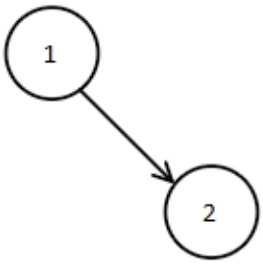
- Insertamos el elemento 1 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
  No es necesaria ninguna rotación.

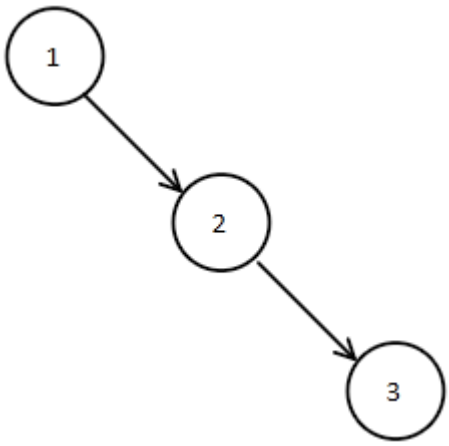
- Insertamos el elemento 2 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 1 = -1
- FE (2) = 0 - 0 = 0
  No es necesaria ninguna rotación.

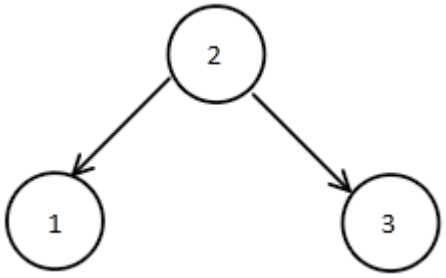
- Insertamos el elemento 3 en el árbol AVL:



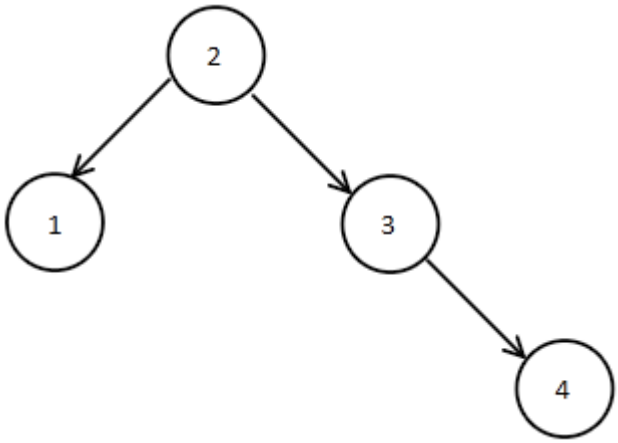
Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 2 = -2
- FE (2) = 0 - 1 = -1
- FE (3) = 0 - 0 = 0

  Rotación derecha – derecha simple sobre el nodo 1



- Insertamos el elemento 4 en el árbol AVL:

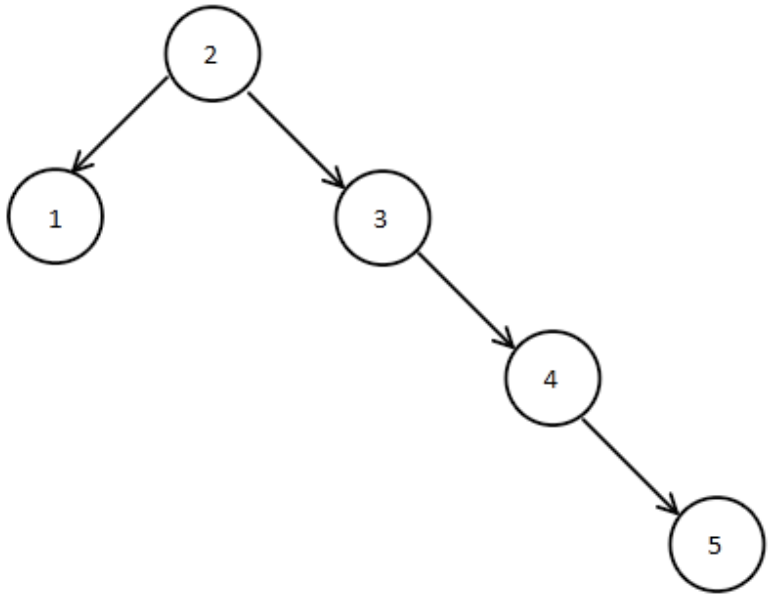


Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 2 = -1
- FE (3) = 0 - 1 = -1
- FE (4) = 0 - 0 = 0

  No es necesaria ninguna rotación.

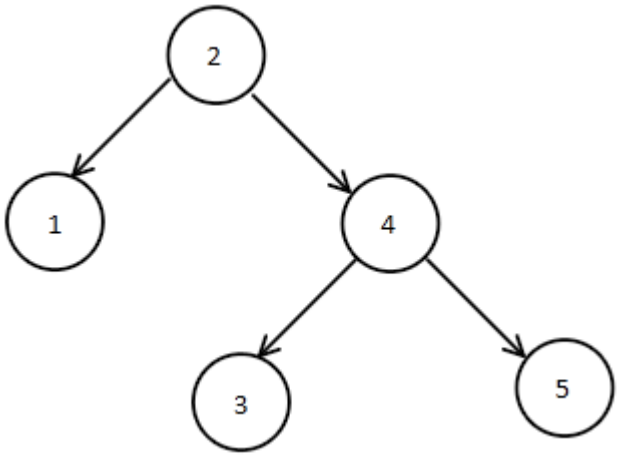
- Insertamos el elemento 5 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 3 = -2
- FE (3) = 0 - 2 = -2
- FE (4) = 0 - 1 = -1
- FE (5) = 0 - 0 = 0

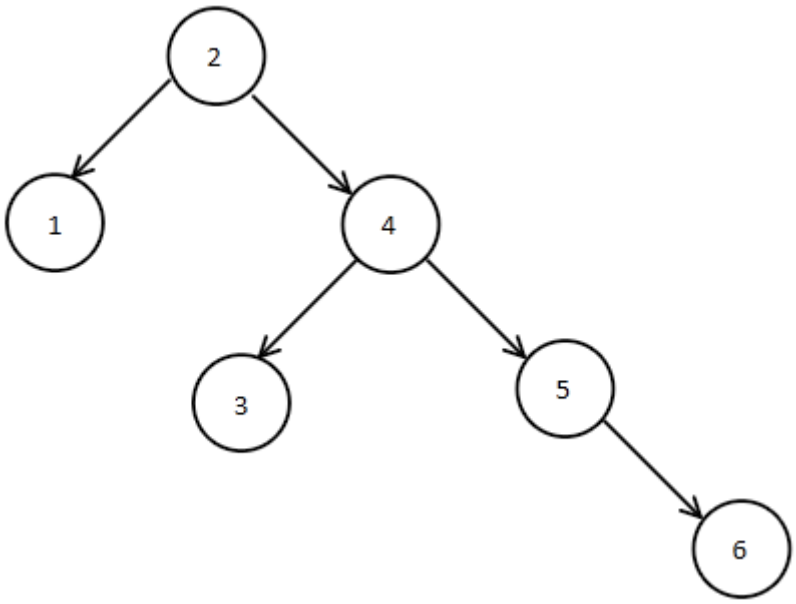
  Rotación derecha – derecha simple sobre el nodo 3



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 2 = -1
- FE (3) = 0 - 0 = 0
- FE (4) = 0 - 1 = -1
- FE (5) = 0 - 0 = 0

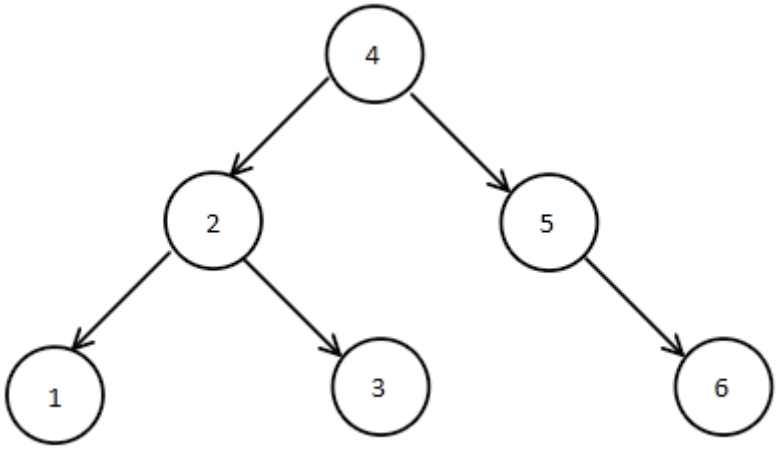
- Insertamos el elemento 6 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 3 = -2
- FE (3) = 0 - 0 = 0
- FE (4) = 1 - 2 = -1
- FE (5) = 0 - 1 = -1
- FE (6) = 0 - 0 = 0

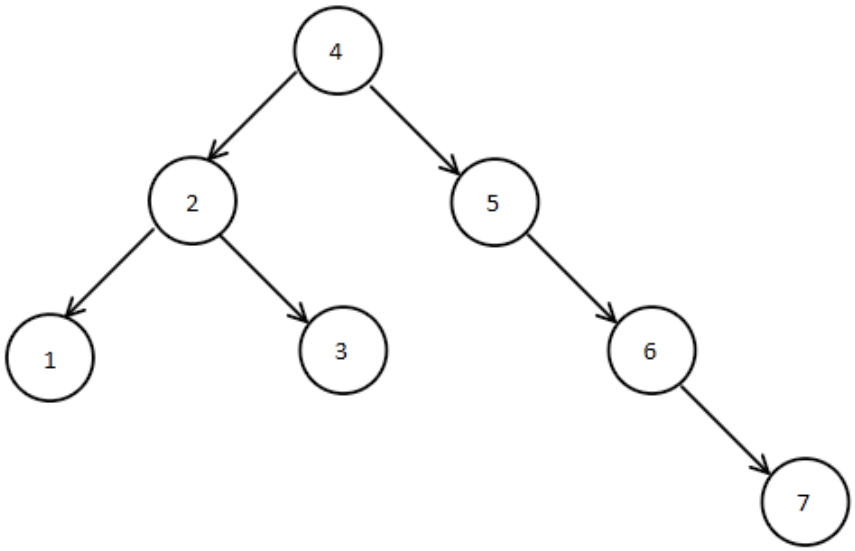
Rotación derecha – derecha simple sobre el nodo 2



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 1 = -1
- FE (6) = 0 - 0 = 0

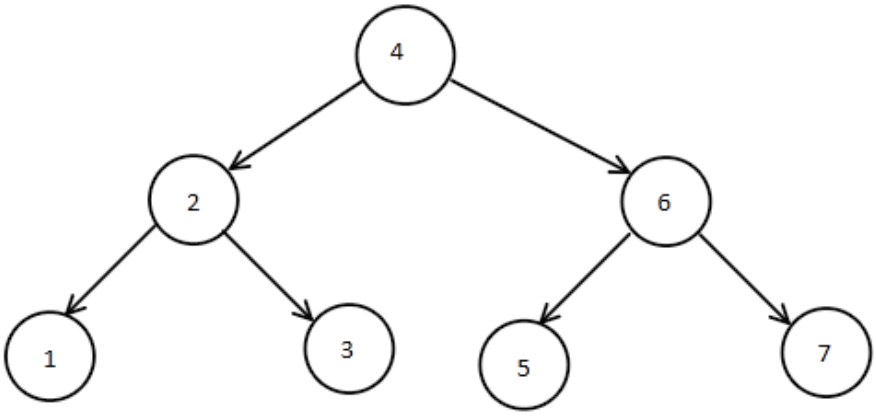
• Insertamos el elemento 7 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 1 - 3 = -2
- FE (5) = 0 - 2 = -2
- FE (6) = 0 - 1 = -1
- FE (7) = 0 - 0 = 0

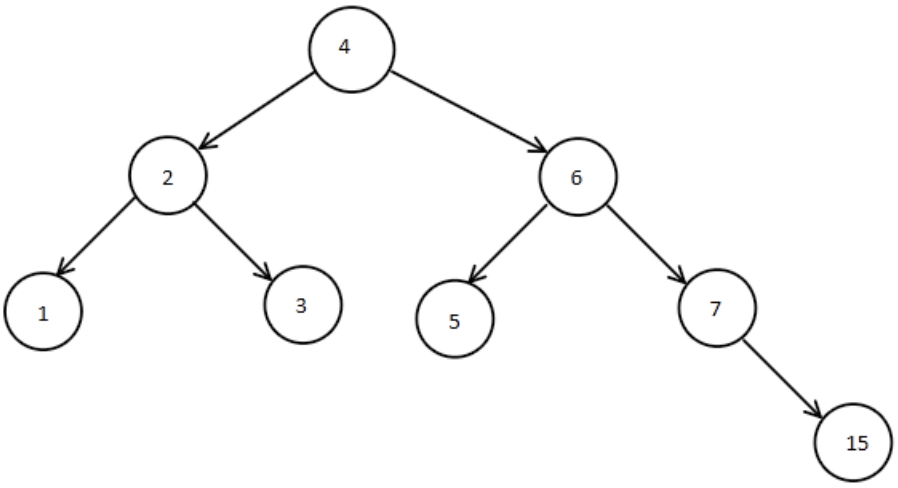
Rotación derecha – derecha simple sobre el nodo 5



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 1 - 2 = -1
- FE (5) = 0 - 0 = 0
- FE (6) = 0 - 1 = -1
- FE (7) = 0 - 0 = 0

- Insertamos el elemento 15 en el árbol AVL:

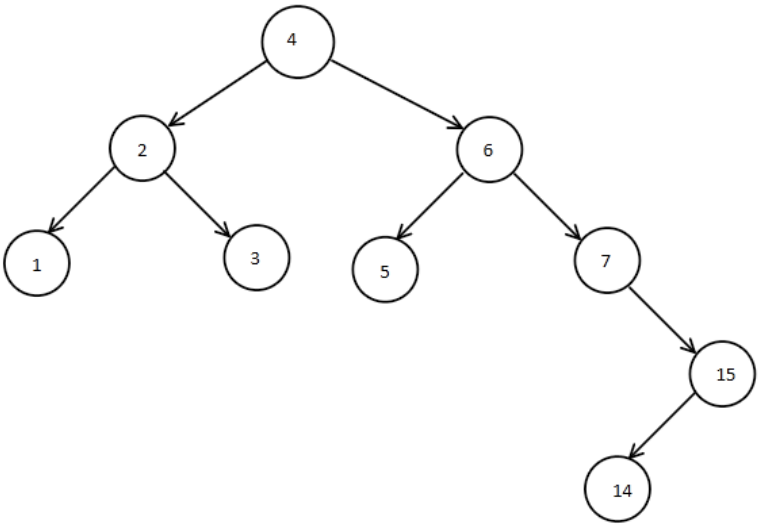


Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 3 = -1
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 2 = -1
- FE (7) = 0 - 1 = -1
- FE (15) = 0 - 0 = 0

  No es necesaria ninguna rotación.

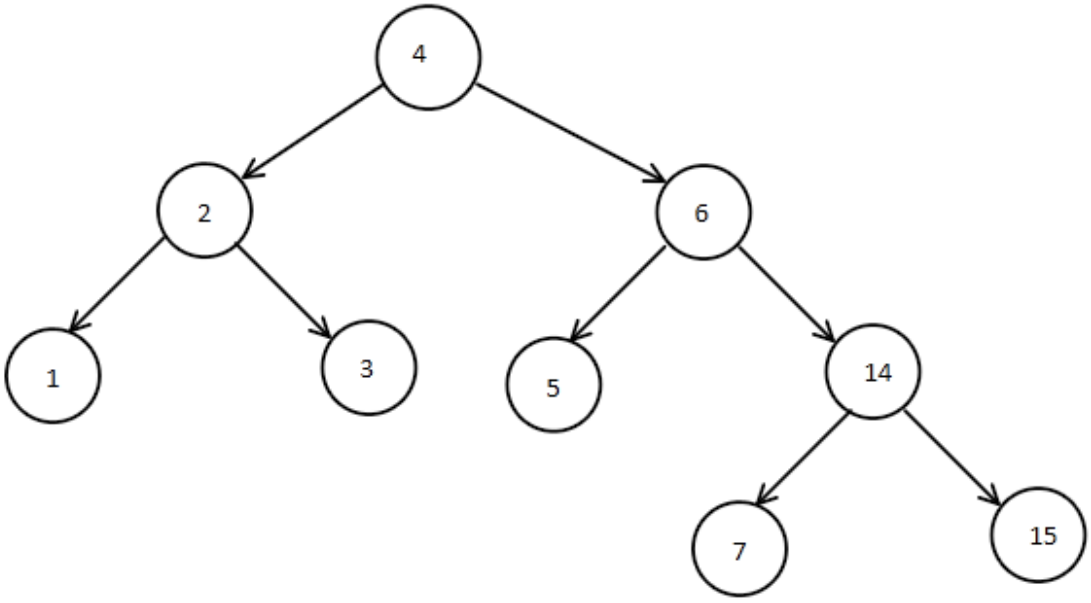
- Insertamos el elemento 14 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 4 = -2
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 3 = -2
- FE (7) = 0 - 2 = -2
- FE (15) = 1 - 0 = 1
- FE (14) = 0 - 0 = 0

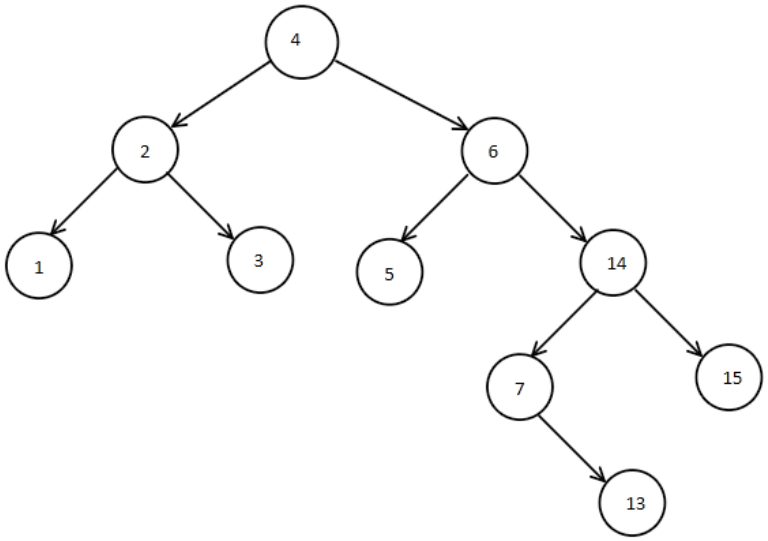
Rotación derecha – derecha simple sobre el nodo 7



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 3 = -1
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 2 = -1
- FE (7) = 0 - 0 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 1 - 1 = 0

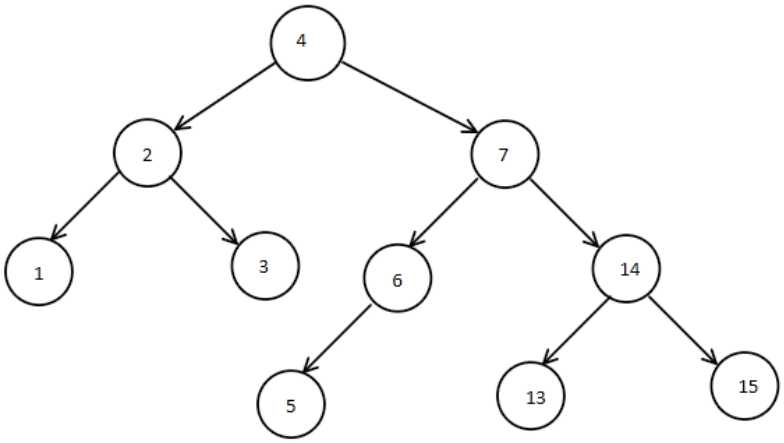
• Insertamos el elemento 13 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 4 = -2
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 3 = -2
- FE (7) = 0 - 1 = -1
- FE (15) = 0 - 0 = 0
- FE (14) = 2 - 1 = 1
- FE (13) = 0 - 0 = 0

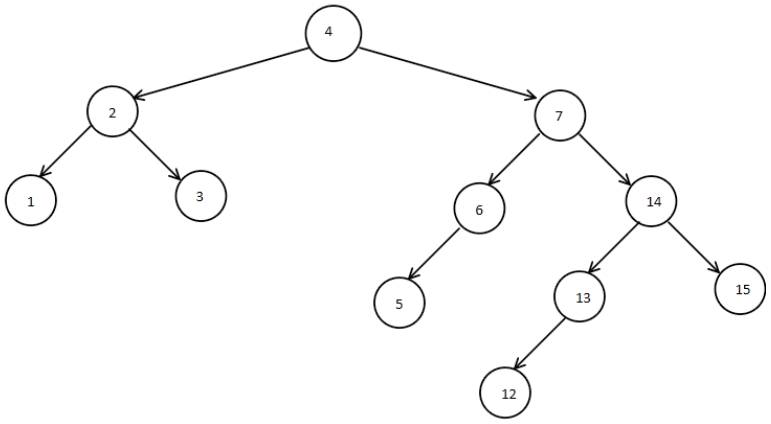
Rotación derecha – izquierda doble sobre el nodo 6



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 3 = -1
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 2 - 2 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 1 - 1 = 0
- FE (13) = 0 - 0 = 0

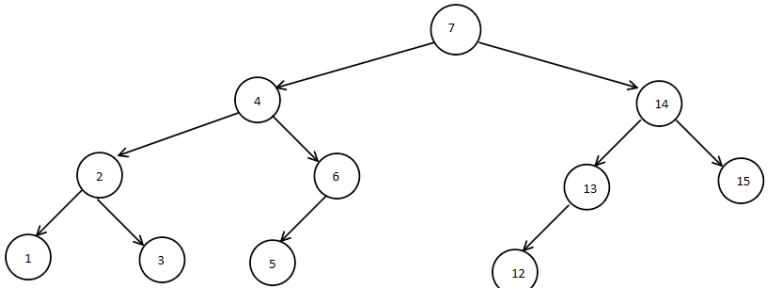
- Insertamos el elemento 12 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 4 = -2
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 2 - 3 = -1
- FE (15) = 0 - 0 = 0
- FE (14) = 2 - 1 = 1
- FE (13) = 1 - 0 = 1
- FE (12) = 0 - 0 = 0

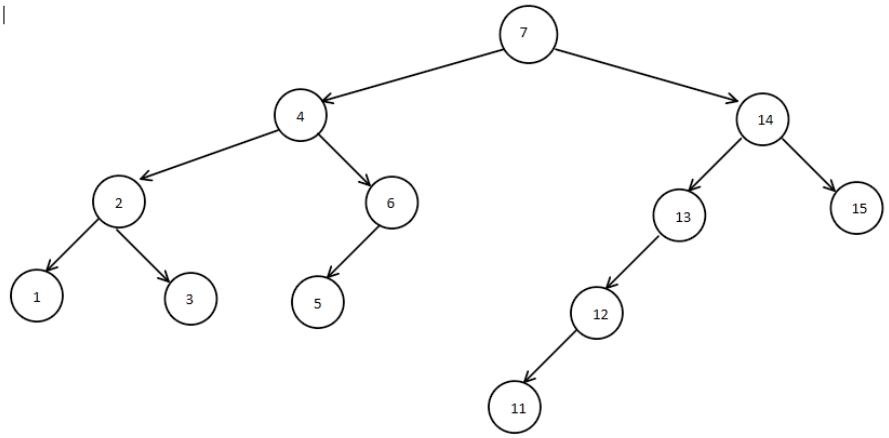
Rotación derecha – derecha simple sobre el nodo 4



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 3 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 2 - 1 = 1
- FE (13) = 1 - 0 = 1
- FE (12) = 0 - 0 = 0

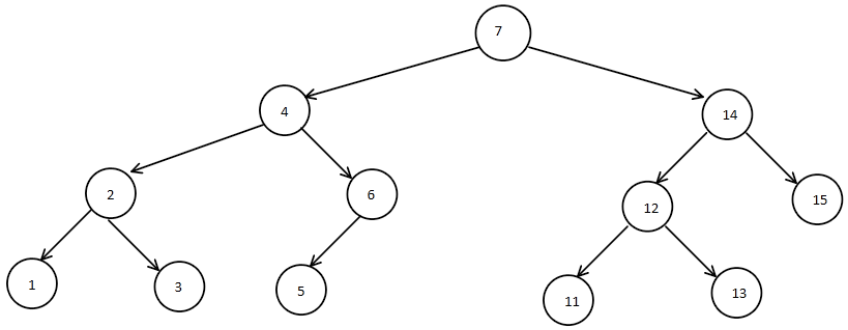
- Insertamos el elemento 11 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 4 = -1
- FE (15) = 0 - 0 = 0
- FE (14) = 3 - 1 = 2
- FE (13) = 2 - 0 = 2
- FE (12) = 1 - 0 = 1
- FE (11) = 0 - 0 = 0

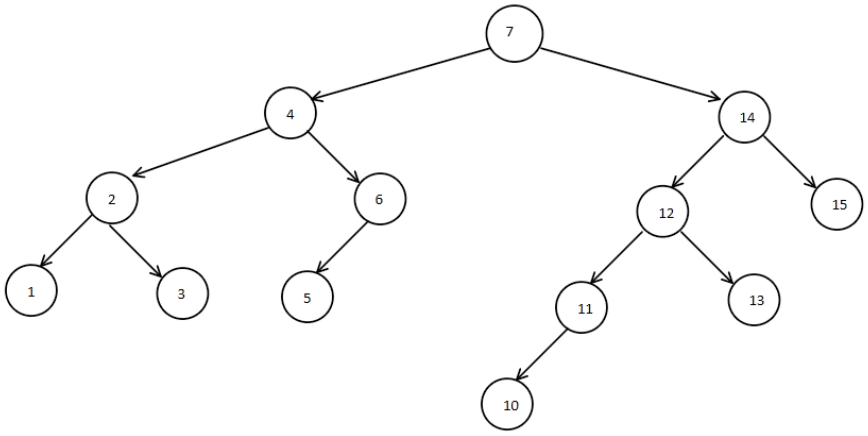
Rotación Izquierda – Izquierda simple sobre el nodo 13



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 3 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 2 - 1 = 1
- FE (13) = 0 - 0 = 0
- FE (12) = 1 - 1 = 0
- FE (11) = 0 - 0 = 0

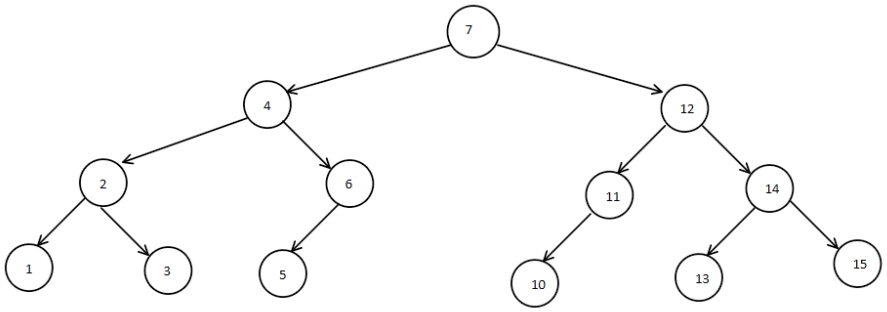
- Insertamos el elemento 10 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 4 = -1
- FE (15) = 0 - 0 = 0
- FE (14) = 3 - 1 = 2
- FE (13) = 0 - 0 = 0
- FE (12) = 2 - 1 = 1
- FE (11) = 1 - 0 = 1
- FE (10) = 0 - 0 = 0

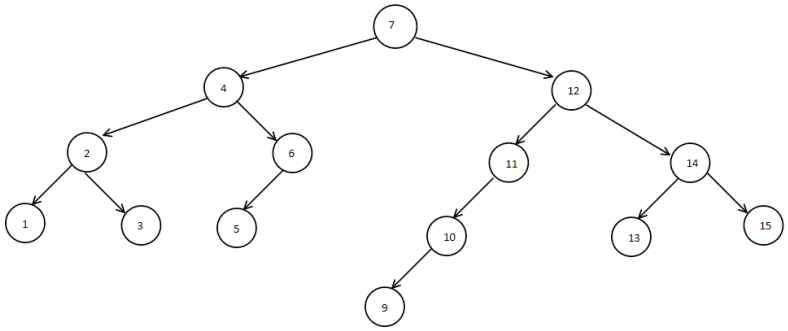
Rotación izquierda – izquierda simple sobre el nodo 14



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 3 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 1 - 1 = 0
- FE (13) = 0 - 0 = 0
- FE (12) = 2 - 2 = 0
- FE (11) = 1 - 0 = 1
- FE (10) = 0 - 0 = 0

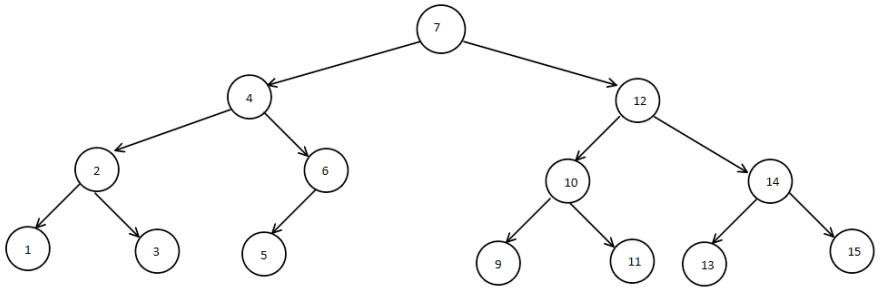
- Insertamos el elemento 9 en el árbol AVL:



Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 4 = -1
- FE (15) = 0 - 0 = 0
- FE (14) = 1 - 1 = 0
- FE (13) = 0 - 0 = 0
- FE (12) = 3 - 2 = 1
- FE (11) = 2 - 0 = 2
- FE (10) = 1 - 0 = 1
- FE (9) = 0 - 0 = 0

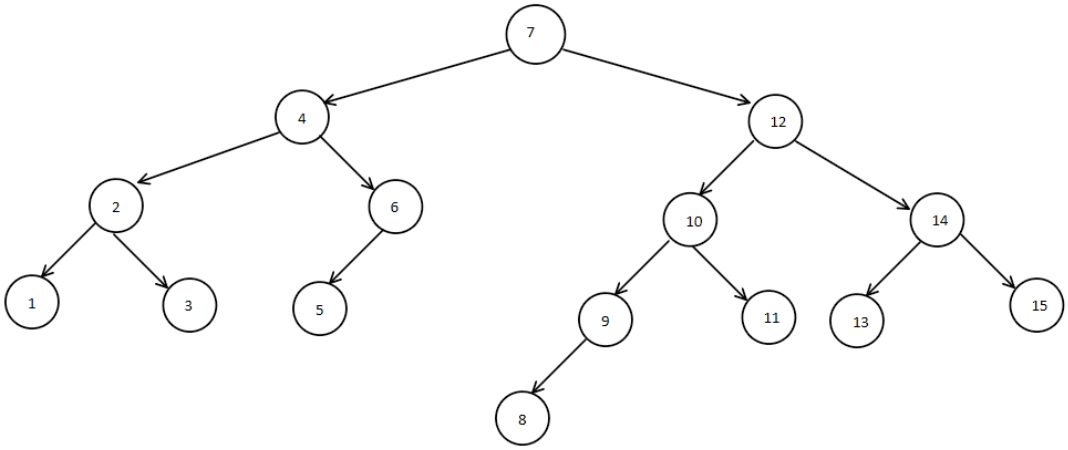
Rotación izquierda – izquierda simple sobre el nodo 11



Comprobación

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 3 = 0
- FE (15) = 0 - 0 = 0
- FE (14) = 1 - 1 = 0
- FE (13) = 0 - 0 = 0
- FE (12) = 2 - 2 = 0
- FE (11) = 0 - 0 = 0
- FE (10) = 1 - 1 = 0
- FE (9) = 0 - 0 = 0

- Insertamos el elemento 8 en el árbol AVL:



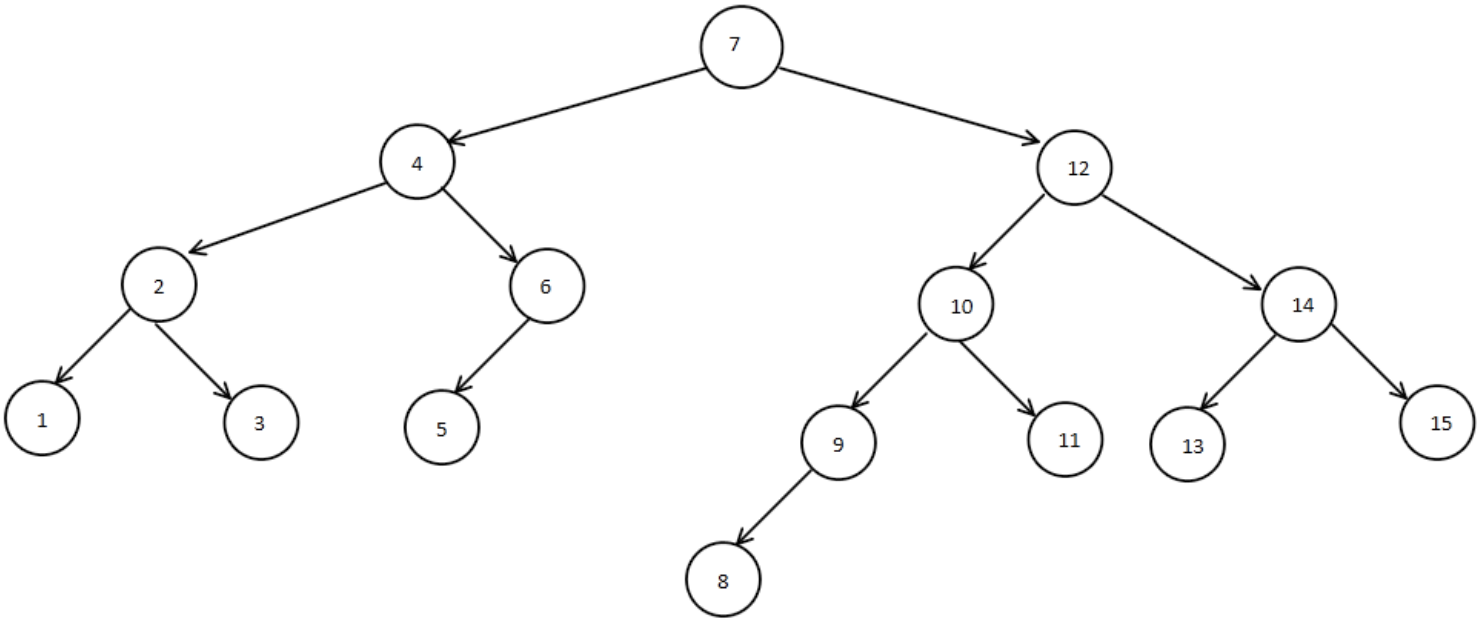
Calculamos el factor de equilibrio (fe).

- FE (1) = 0 - 0 = 0
- FE (2) = 1 - 1 = 0
- FE (3) = 0 - 0 = 0
- FE (4) = 2 - 2 = 0
- FE (5) = 0 - 0 = 0
- FE (6) = 1 - 0 = 1
- FE (7) = 3 - 4 = -1
- FE (15) = 0 - 0 = 0

  No es necesaria ninguna rotación.

- FE (14) = 1 - 1 = 0
- FE (13) = 0 - 0 = 0
- FE (12) = 3 - 2 = -1
- FE (11) = 0 - 0 = 0
- FE (10) = 2 - 1 = 1
- FE (9) = 1 - 0 = 1
- FE (8) = 0 - 0 = 0

Árbol AVL tras insertar todos los elementos



EJERCICIO 2

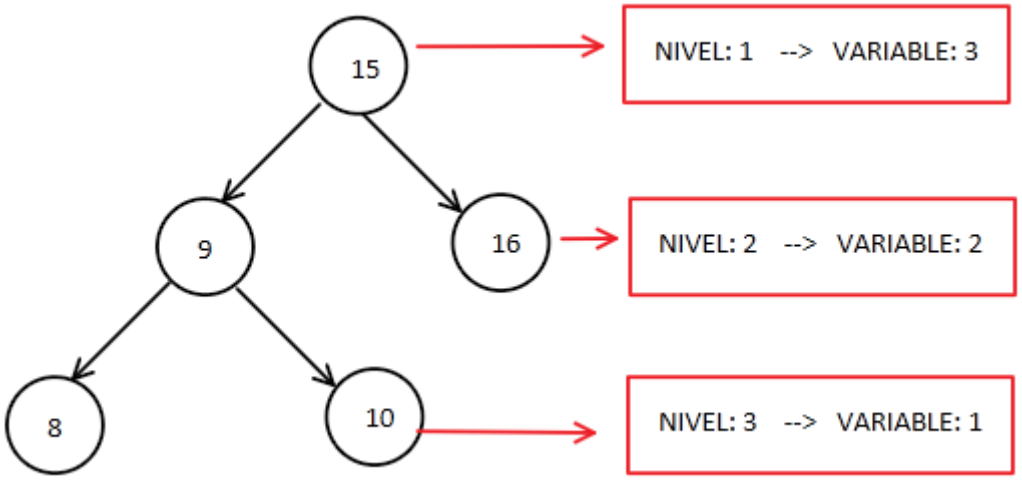
Diseñar(pseudocódigo) e implementar en Java una función que reciba como entrada un árbol binario de letras, y que escriba por pantalla los caracteres almacenados en el último nivel. Utilizar para ello las operaciones del TAD ArbolBin visto en la asignatura y la implementación proporcionada. Pueden codificarse las funciones auxiliares que se deseen. Calcular de manera razonada su eficiencia, justificando las complejidades de las operaciones y funciones utilizadas.

Para completar este ejercicio, he creado dos funciones:

- **Función height**: recibe como parámetro el árbol. Es una función que devuelve la altura del árbol, de esta manera podremos calcular cual es el último nivel. Vamos a recorrer el árbol actualizando una variable obteniendo el máximo entre el subárbol izquierdo y derecho de cada nodo.
- **Función ultimoNivel**: recibe como parámetros el árbol y el nivel (calculado en la función height). Es una función que recorre recursivamente el árbol, y por cada llamada recursiva reduce en uno la variable nivel. Cuando esta variable llega al valor 1, significa que estamos en el último nivel, por lo que se imprime el nodo en el que nos encontramos.

Idea de resolución

Cuando calculamos la altura de un árbol, nos da como resultado el número de niveles que tenemos. Con esta información, mi idea es recorrer el árbol de forma recursiva, disminuyendo una variable (con valor inicial = altura del árbol). Cuando esta variable tenga el valor 1, significara que estamos en el último nivel del árbol, donde mostraremos por consola el resultado.



Para calcular la altura del árbol, he creado una función que recorre el árbol recursivamente comparando la altura de cada subárbol (función Math.max de java). El parámetro que devuelve esta función es el que le paso a la función que imprime el ultimo nivel.

```java
public static void main(String[] args) {
    ArbolBin<String> g = new ArbolBin<String>(new ArbolBin<String>(),"J",new ArbolBin<String>());
    ArbolBin<String> d = new ArbolBin<String>(g,"A",new ArbolBin<String>());
    ArbolBin<String> f = new ArbolBin<String>(new ArbolBin<String>(),"B",new ArbolBin<String>());
    ArbolBin<String> e = new ArbolBin<String>(new ArbolBin<String>(),"C",f);
    ArbolBin<String> b = new ArbolBin<String>(d,"D",new ArbolBin<String>());
    ArbolBin<String> c = new ArbolBin<String>(e,"E",new ArbolBin<String>());

    ArbolBin<String> a = new ArbolBin<String>(b,"L",c);

    a.dibujar(1);


    int ultiLevel = AlgoritmoEscribeCaracteresPerez.height(a);

    System.out.println("Nodos del ultimo nivel (nivel: "+ultiLevel+"):");
    AlgoritmoEscribeCaracteresPerez.ultimoNivel(a, ultiLevel);

}
```

Pseudocódigo:

función enteros height (ArbolBin: ArbolBin<T>)
    int altura inicializada a 0
    si ArbolBin no está vacío
        si el hijo izquierdo de ArbolBin no está vacío
            altura = máximo entre altura y función recursiva height(hijo izquierdo de ArbolBin)

        si el hijo derecho de ArbolBin no está vacío
            altura = máximo entre altura y función recursiva height(hijo derecho de ArbolBin)

        altura++
    fin si
    devolver altura

función ultimoNivel(ArbolBin: ArbolBin<T>, nivel: entero)
    si nivel es igual a 1
        imprimir elemento del nodo en el que estemos
    fin si

    si el hijo izquierdo del ArbolBin no está vacío
        función recursiva ultimoNivel(hijo izquierdo de ArbolBin, nivel-1)

    si el hijo derecho del ArbolBin no está vacío
        función recursiva ultimoNivel(hijo derecho de ArbolBin, nivel-1)

Códigos utilizados:

```
public static int height(ArbolBin<String> arbolBin) {
	int altura = 0;

	if(!arbolBin.esVacio()) {
		if (!arbolBin.hijoIzquierdo().esVacio()) {
			altura = Math.max(altura,
			height(arbolBin.hijoIzquierdo()));
		}
		if(!arbolBin.hijoDerecho().esVacio()) {
			altura = Math.max(altura,
			height(arbolBin.hijoDerecho()));
		}
		altura++;
	}
	return altura;
}

public static void ultimoNivel(ArbolBin<String> arbolBin, int nivel) {
	if(nivel == 1) {
		System.out.print(arbolBin.raiz() + " - ");
	}
	if(!arbolBin.hijoIzquierdo().esVacio()) {
		ultimoNivel(arbolBin.hijoIzquierdo(), nivel-1);
	}
	if(!arbolBin.hijoDerecho().esVacio()) {
		ultimoNivel(arbolBin.hijoDerecho(), nivel-1);
	}
}
```

Tiempo y orden de ejecución:

```
public static void ultimoNivel(ArbolBin<String> arbolBin, int nivel) {
    if(nivel == 1) {                                          A
        System.out.print(arbolBin.raiz() + " - ");
    }
    if(!arbolBin.hijoIzquierdo().esVacio()) {
        ultimoNivel(arbolBin.hijoIzquierdo(), nivel-1);       B
    }
    if(!arbolBin.hijoDerecho().esVacio()) {
        ultimoNivel(arbolBin.hijoDerecho(), nivel-1);
    }                                                         C
}
```

Caso base $(C_0 n^{k_0})$

$T(A) = 1 + (1 + 1) = 3$

$C_0 = 3 \qquad n^{k_0} = 1 \rightarrow k_0 = 1$

Caso general $(a \cdot T(n/b) + C_1 n^{k_1})$

$T_{total} = T(B) + T(C) \rightarrow T(B) = T(C)$

$$T_{Total} = \begin{cases} P.N.R \\ P.R \end{cases}$$

$P.N.R. \rightarrow 2 \cdot (1 + 1) = 4 \rightarrow n^{k_1} = 1 \rightarrow k_1 = 0$

$P.R. \rightarrow a = 2 \qquad b = 2$

$T_{Total} = 2 \cdot T(n/2) + 4$

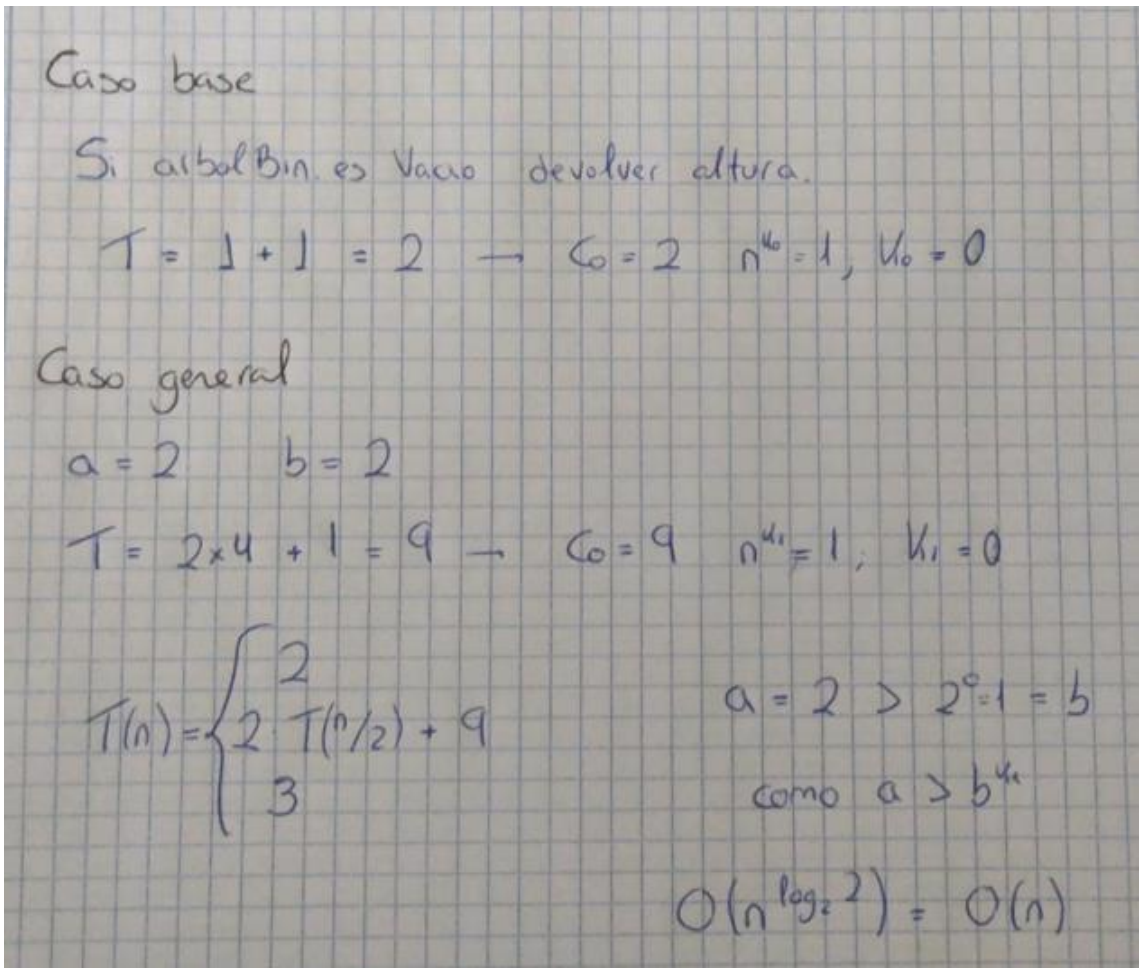
$$T(n) = \begin{cases} 3 \\ 2 \cdot T(n/2) + 4 \end{cases}$$

$a \leftrightarrow b^{k_0} \rightarrow 2 \leftrightarrow 2^0$

como $a = 2 > 1 = b^{k_0}$:

$O(n^{\log_2 2}) = O(n)$

```java
public static int height(ArbolBin<String> arbolBin) {
    int altura = 0;

    if(!arbolBin.esVacio()) {

        if (!arbolBin.hijoIzquierdo().esVacio()) {
            altura = Math.max(altura,  height(arbolBin.hijoIzquierdo()));
        }

        if(!arbolBin.hijoDerecho().esVacio()) {
            altura = Math.max(altura,  height(arbolBin.hijoDerecho()));
        }
        altura++;
    }
    return altura;
}
```



EJERCICIO 3

Código de la función implementada

```
//ALGORITMO KRUSKAL
public Lista <Par<Clave>> AlgoritmoKruskalAR (Grafo <String, String, Integer>
grafo){

	Lista<Clave> visitados = new Lista<Clave>();
	Lista <Par<Clave>> sol = new Lista <Par<Clave>>();

	int[][] arista = new int[vertices.longitud()][vertices.longitud()];

	//crear matriz de adyacencia
	for(int i = 1; i <= vertices.longitud(); i++) {
		for(int j = 1; j <= vertices.longitud(); j++) {
			arista[i-1][j-1] =
			costeArista2(vertices.consultar(i).clave,vertices.consulta
			r(j).clave);
		}
	}

	int menor = 10000;
	int costeTotal = 0;
	int verticesVistos = 1;

	while(vertices.longitud() > verticesVistos) {
		for(int i = 1; i <= vertices.longitud(); i++) {
		  for(int j = 1; j <= vertices.longitud(); j++) {
			if(arista[i-1][j-1] < menor && arista[i-1][j-1] != 0) {
				if(visitados.longitud() != 0) {
					menor = arista[i-1][j-1];
				}else {

				if(comprobarVisitados(visitados,
				vertices.consultar(i).clave,
				vertices.consultar(j).clave)) {
					menor = arista[i-1][j-1];
						}
					}
				}
			}
		}


		for (int i = 1; i <= vertices.longitud(); i++) {
		 for (int j = 1; j <= vertices.longitud(); j++) {
			if(arista[i-1][j-1] == menor) {
				arista[i-1][j-1] = 0;
				visitados.insertar(1, vertices.consultar(i).clave);
				visitados.insertar(1, vertices.consultar(j).clave);
				sol.insertar(1, new
				Par(vertices.consultar(i).clave,vertices.consultar(
				j).clave));
				menor = 100000;
				verticesVistos++;
				costeTotal = costeTotal +
				costeArista2(vertices.consultar(i).clave,vertices.c
				onsultar(j).clave);
```

```
                    }
                }
            }
        }

for (int i = 1; i <= sol.longitud(); i++) {
        System.out.println(sol.consultar(i).getOrigen()+ " --> " +
        sol.consultar(i).getDestino());
}

System.out.println("Coste total de las aristas = "+costeTotal);
return sol;

}
```

Funciones adicionales:

- `private int costeArista2(Clave o, Clave d)`
  En esta función, vamos a conseguir obtener el valor del coste que tiene una arista como si fuera un entero. Esto nos permitirá hacer las comparaciones necesarias para encontrar el menor elemento.

- `private boolean comprobarVisitados(Lista<Clave> visitados, Clave clave, Clave clave2)`
  En esta función vamos a comprobar, si con la arista más pequeña que obtengamos, formamos un ciclo con alguno de los vértices ya visitados.

- `private Lista<Clave> listaVerticesUnidos(Clave v)`
  En esta función, obtendremos todos los sucesores y predecesores de un vértice determinado. Esto nos ayudara en la función *"comprobarVisitados"*, sabiendo si ya hemos visitado algún sucesor/predecesor del vértice. Como tenemos un grafo dirigido, necesitamos esta función para saber los vértices con los que se relacionaría si fuera un grafo no dirigido.

Idea de resolución

Para poder completar este algoritmo, empezaremos creado dos listas, una que se encargara de almacenar únicamente la clave de los elementos que vayamos visitando del grafo, y otra que almacenara el par de claves que cumplen las condiciones del algoritmo y que se mostraran en la solución.

Para poder buscar los costes de aristas más pequeños, crearemos la matriz de adyacencia del grafo correspondiente.

Una vez realizados estos pasos, entraremos a un bucle que no saldremos hasta que hayamos visitado todos los vértices del grafo inicial. Dentro de este bucle tenemos los siguientes pasos:

Buscar cual es la arista que tiene el valor más pequeño, y que si al juntar los vértices correspondientes con la solución estos no formarían un ciclo. En caso de que no se forme un ciclo, obtendremos el valor más pequeño de la arista del grafo.

Buscar cuales son los vértices correspondientes a la arista obtenida. Cambiamos el valor de la arista en la matriz de adyacencia, para no volver a considerarla y poder buscar la siguiente más pequeña. Después insertamos los vértices correspondientes en la lista de vértices visitados, y esos dos vértices, forman un par de claves, que se introduce en la lista que mostrar la solución. Al mismo tiempo, almacenamos en una variable el coste de la arista, que se sumara al resto de aristas pertenecientes a la lista de pares de clave.

Por último, para mostrar la solución, recorremos la lista de pares, mostrando el vértice origen y destino de cada para, y al final, la suma del coste de todas las aristas visitadas.

Tiempo y orden de complejidad

F

```
public Lista <Par<Clave>> AlgoritmoKruskalAR (Grafo <String, String, Integer> grafo){

    Lista<Clave> visitados = new Lista<Clave>();
    Lista <Par<Clave>> sol = new Lista <Par<Clave>>();

    int[][] arista = new int[vertices.longitud()][vertices.longitud()];

    for(int i = 1; i <= vertices.longitud(); i++) {
        for(int j = 1; j <= vertices.longitud(); j++) {
            arista[i-1][j-1] = costeArista2(vertices.consultar(i).clave,vertices.consultar(j).clave);
        }
    }

    int menor = 10000;
    int costeTotal = 0;
    int verticesVistos = 1;

    while(vertices.longitud() > verticesVistos) {

        for(int i = 1; i <= vertices.longitud(); i++) {
            for(int j = 1; j <= vertices.longitud(); j++) {
                if(arista[i-1][j-1] < menor && arista[i-1][j-1] != 0) {
                    if(visitados.longitud() != 0) {
                        menor = arista[i-1][j-1];
                    }else {
                        if(comprobarVisitados(visitados, vertices.consultar(i).clave, vertices.consultar(j).clave)) {
                            menor = arista[i-1][j-1];
                        }
                    }
                }
            }
        }


        for (int i = 1; i <= vertices.longitud(); i++) {
            for (int j = 1; j <= vertices.longitud(); j++) {
                if(arista[i-1][j-1] == menor) {

                    arista[i-1][j-1] = 0;
                    visitados.insertar(1, vertices.consultar(i).clave);
                    visitados.insertar(1, vertices.consultar(j).clave);
                    sol.insertar(1, new Par(vertices.consultar(i).clave,vertices.consultar(j).clave));
                    menor = 100000;
                    verticesVistos++;
                    costeTotal = costeTotal + costeArista2(vertices.consultar(i).clave,vertices.consultar(j).clave);
                }
            }
        }
    }

    for (int i = 1; i <= sol.longitud(); i++) {
            System.out.println(sol.consultar(i).getOrigen()+ " --> " + sol.consultar(i).getDestino());
    }
    System.out.println("Coste total de las aristas = "+costeTotal);
    return sol;

}
```

A

B

C

D

E

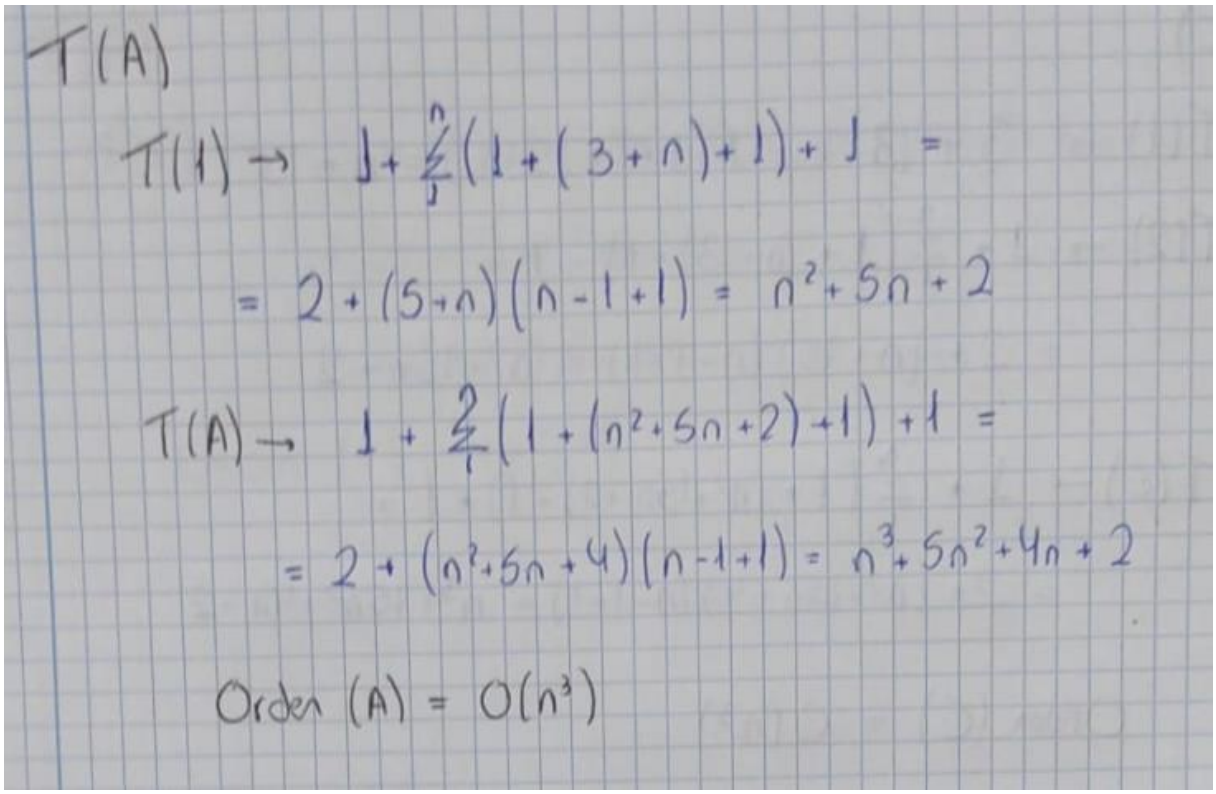
FUNCION	ORDEN
costeArista2	O(n)
comprobarVisitados	O(n)
listaVerticesUnidos	$O(n^2)$

Tiempo y orden de A:

- Tiempo → $n^3 + 5n^2 + 4n + 2$
- Orden → $O(n^3)$

```
for(int i = 1; i <= vertices.longitud(); i++) {
    for(int j = 1; j <= vertices.longitud(); j++) {
        arista[i-1][j-1] = costeArista2(vertices.consultar(i).clave,vertices.consultar(j).clave);
    }
}
```

A	1

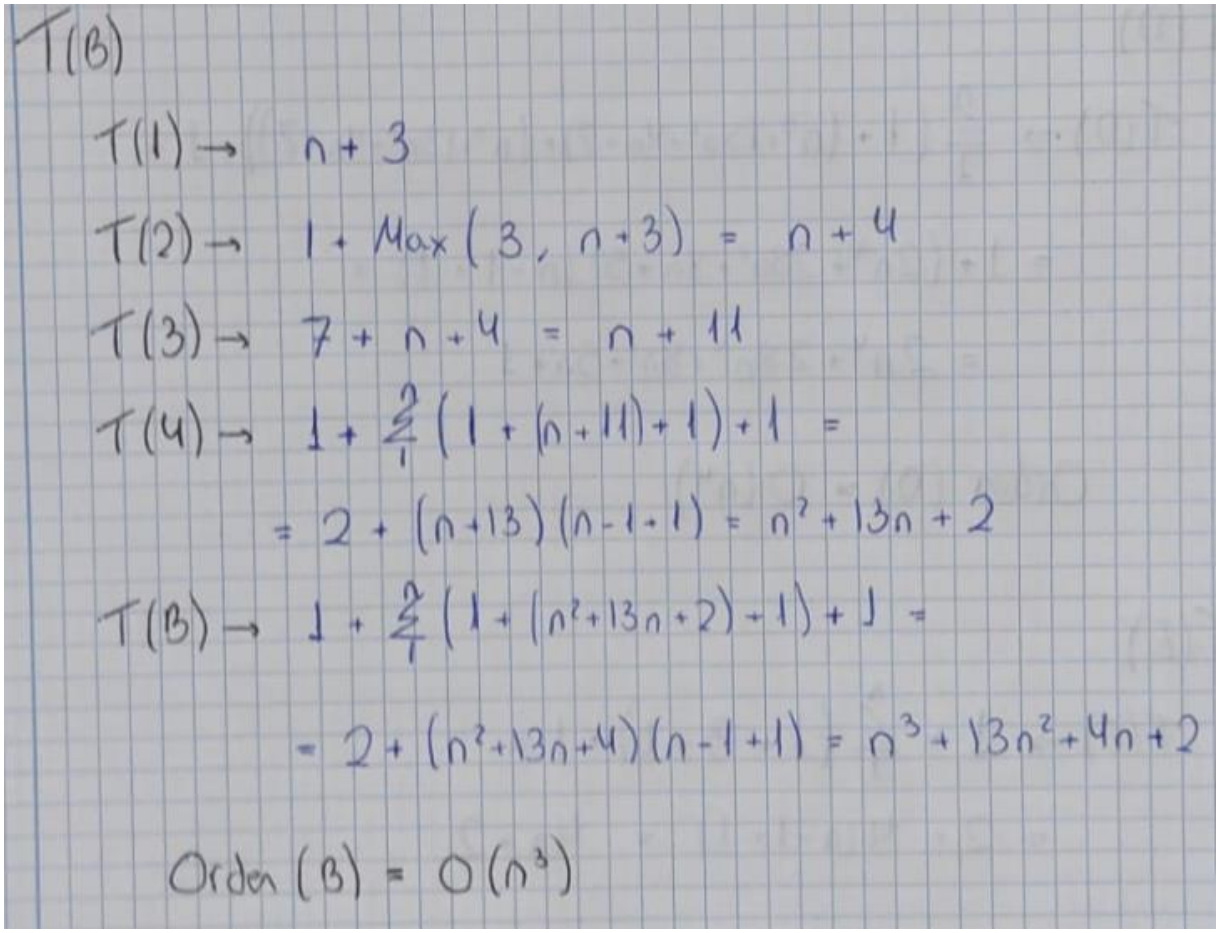


Tiempo y orden de B:

- Tiempo → $n^3 + 13n^2 + 4n + 2$
- Orden → $O(n^3)$

```
for(int i = 1; i <= vertices.longitud(); i++) {
    for(int j = 1; j <= vertices.longitud(); j++) {
        if(arista[i-1][j-1] < menor && arista[i-1][j-1] != 0) {
            if(visitados.longitud() != 0) {
                menor = arista[i-1][j-1];
            }else {
                if(comprobarVisitados(visitados, vertices.consultar(i).clave, vertices.consultar(j).clave)) {
                    menor = arista[i-1][j-1];
                }
            }
        }
    }
}
```

B 4 3 2 1

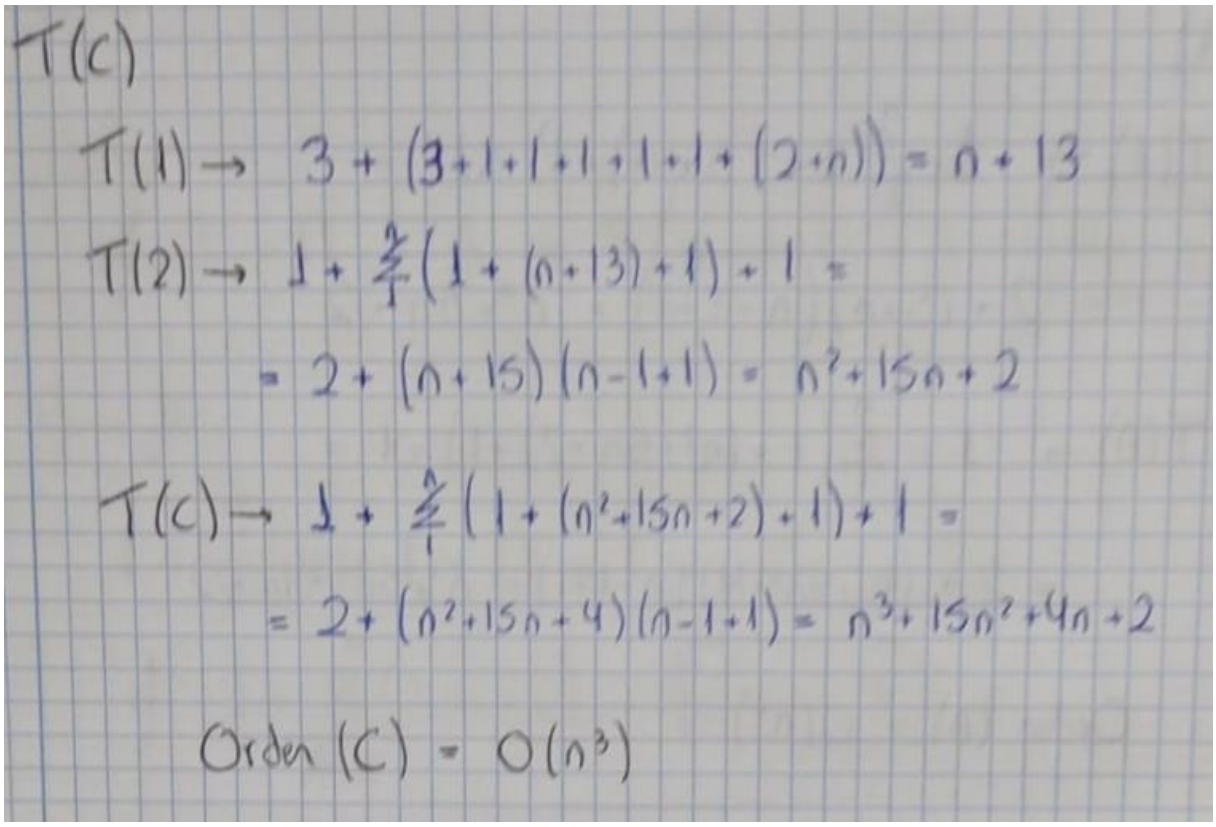


Tiempo y orden de C:

- Tiempo → $n^3 + 15n^2 + 4n + 2$
- Orden → $O(n^3)$

```
for (int i = 1; i <= vertices.longitud(); i++) {
    for (int j = 1; j <= vertices.longitud(); j++) {
        if(arista[i-1][j-1] == menor) {

            arista[i-1][j-1] = 0;
            visitados.insertar(1, vertices.consultar(i).clave);
            visitados.insertar(1, vertices.consultar(j).clave);
            sol.insertar(1, new Par(vertices.consultar(i).clave,vertices.consultar(j).clave));
            menor = 100000;
            verticesVistos++;
            costeTotal = costeTotal + costeArista2(vertices.consultar(i).clave,vertices.consultar(j).clave);
        }
    }
}
```

C	2	1



Tiempo y orden de D:

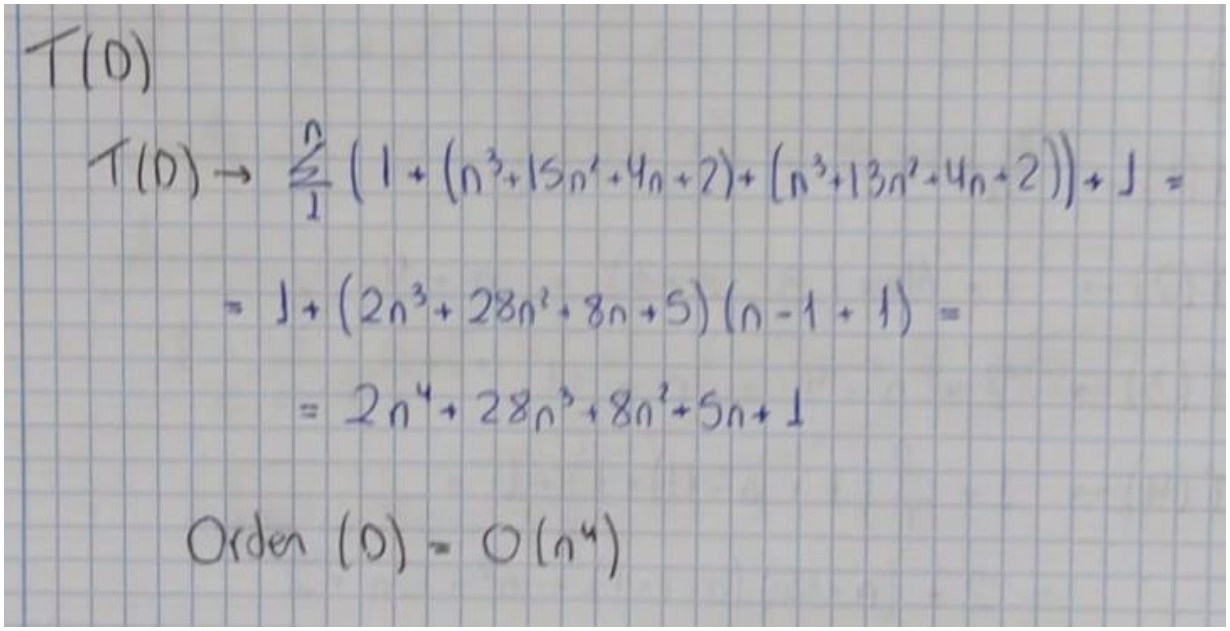
- Tiempo → $2n^4 + 28n^3 + 8n^2 + 5n + 1$
- Orden → $O(n^4)$

```
while(vertices.longitud() > verticesVistos) {
    for(int i = 1; i <= vertices.longitud(); i++) {
        for (int i = 1; i <= vertices.longitud(); i++) {
        }
```

B

D

C

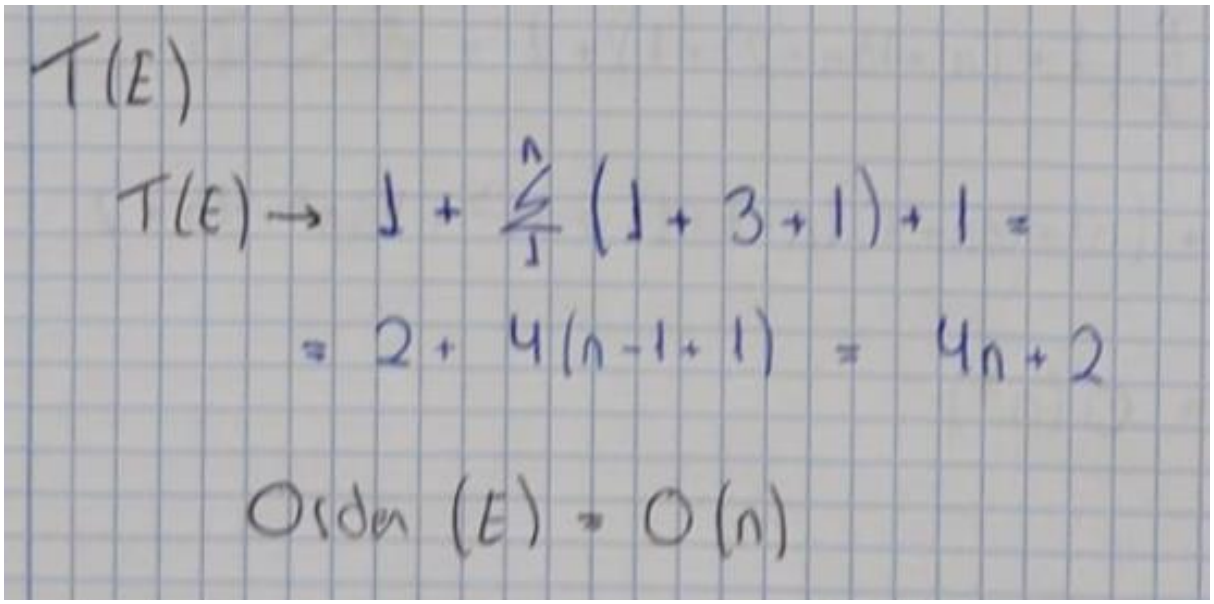


Tiempo y orden de E:

- Tiempo → 4n + 2
- Orden → O(n)

E

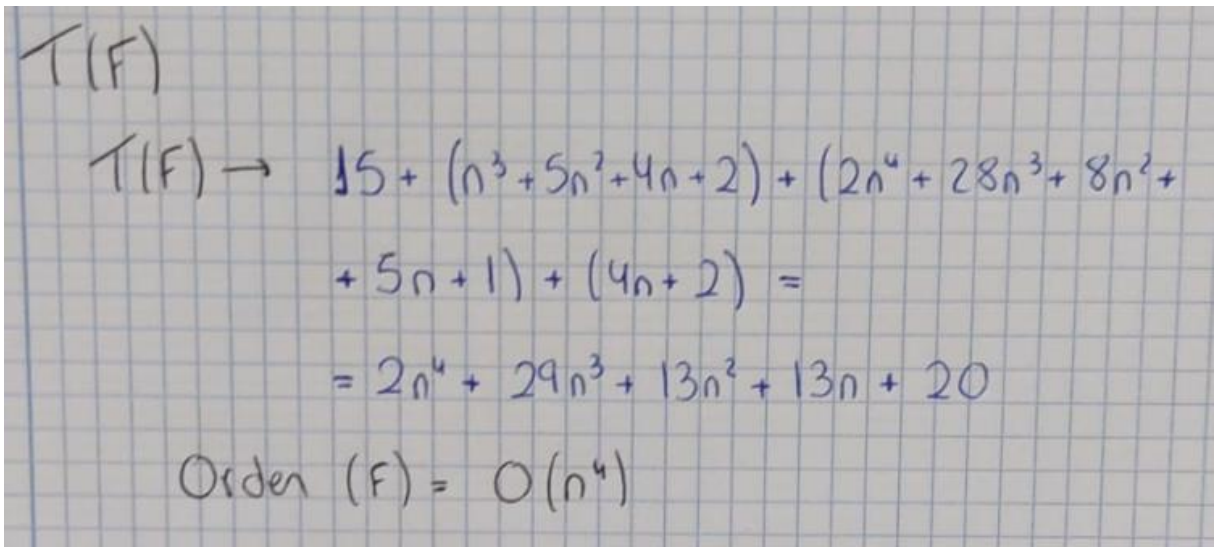
```
for (int i = 1; i <= sol.longitud(); i++) {
        System.out.println(sol.consultar(i).getOrigen()+ " --> " + sol.consultar(i).getDestino());
}
```



Tiempo y orden de F:

- Tiempo → $2n^4 + 29n^3 + 13n^2 + 13n + 20$
- Orden → $O(n^4)$

```
public Lista <Par<Clave>> AlgoritmoKruskalAR (Grafo <String, String, Integer> grafo){

    Lista<Clave> visitados = new Lista<Clave>();
    Lista <Par<Clave>> sol = new Lista <Par<Clave>>();

    int[][] arista = new int[vertices.longitud()][vertices.longitud()];

A   for(int i = 1; i <= vertices.longitud(); i++) {

    int menor = 10000;
    int costeTotal = 0;
F   int verticesVistos = 1;

    while(vertices.longitud() > verticesVistos) {

        for(int i = 1; i <= vertices.longitud(); i++) {
D

        for (int i = 1; i <= vertices.longitud(); i++) {
    }

E   for (int i = 1; i <= sol.longitud(); i++) {

    System.out.println("Coste total de las aristas = "+costeTotal);
    return sol;

}
```



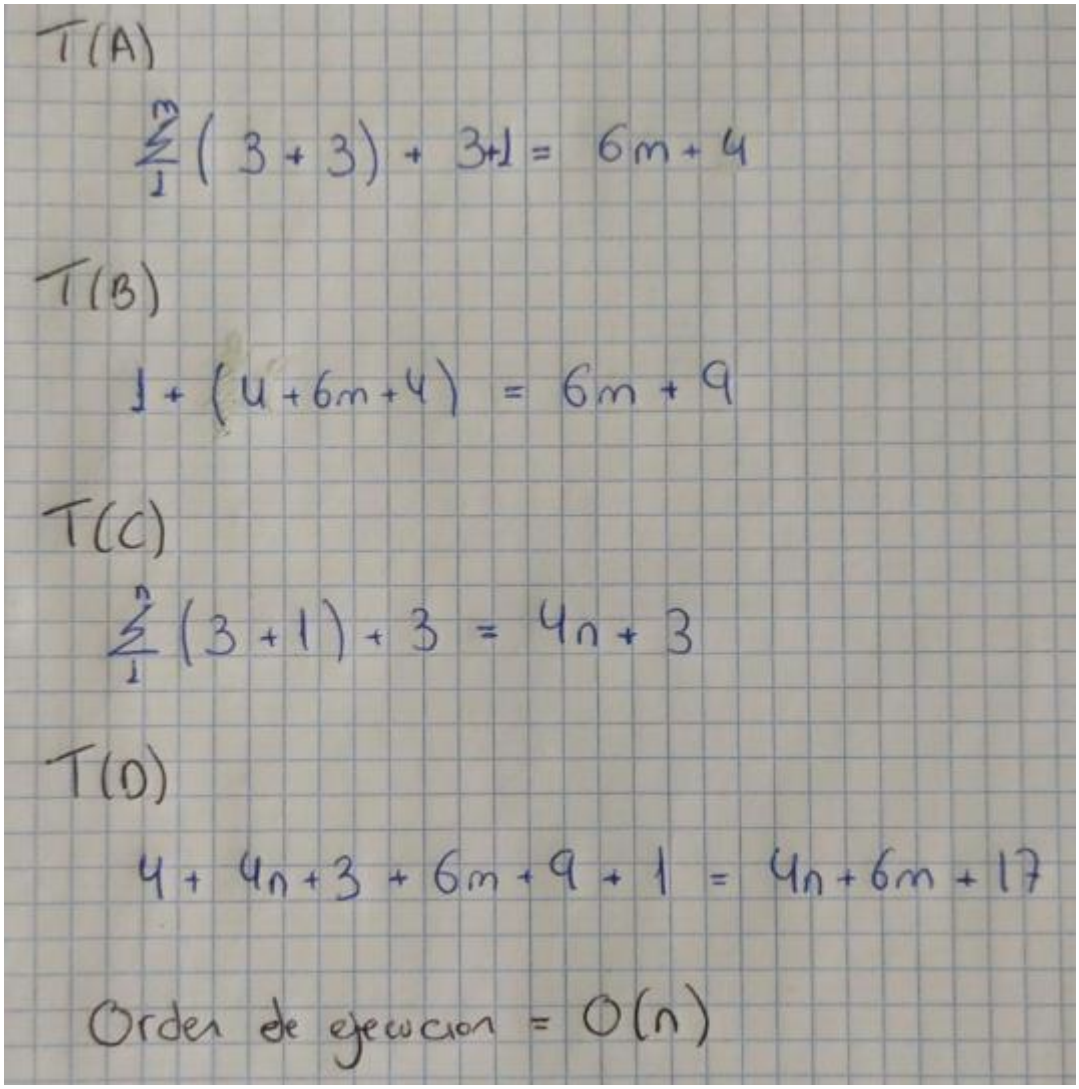
```java
private int costeArista2(Clave o, Clave d) {
    int i = 1;
    int coste = 0;

    while (i <= vertices.longitud() && !vertices.consultar(i).clave.equals(o))
        i++;

    if (i <= vertices.longitud()) {
        int j = 1;
        boolean aristaEncontrada = false;
        while (!aristaEncontrada && j <= aristas.consultar(i).longitud()) {
            if (aristas.consultar(i).consultar(j).destino.clave.equals(d)) {
                coste = (int) aristas.consultar(i).consultar(j).coste;
                aristaEncontrada = true;
            }
            j++;
        }
    }

    return coste;
}
```

C

D

B

A



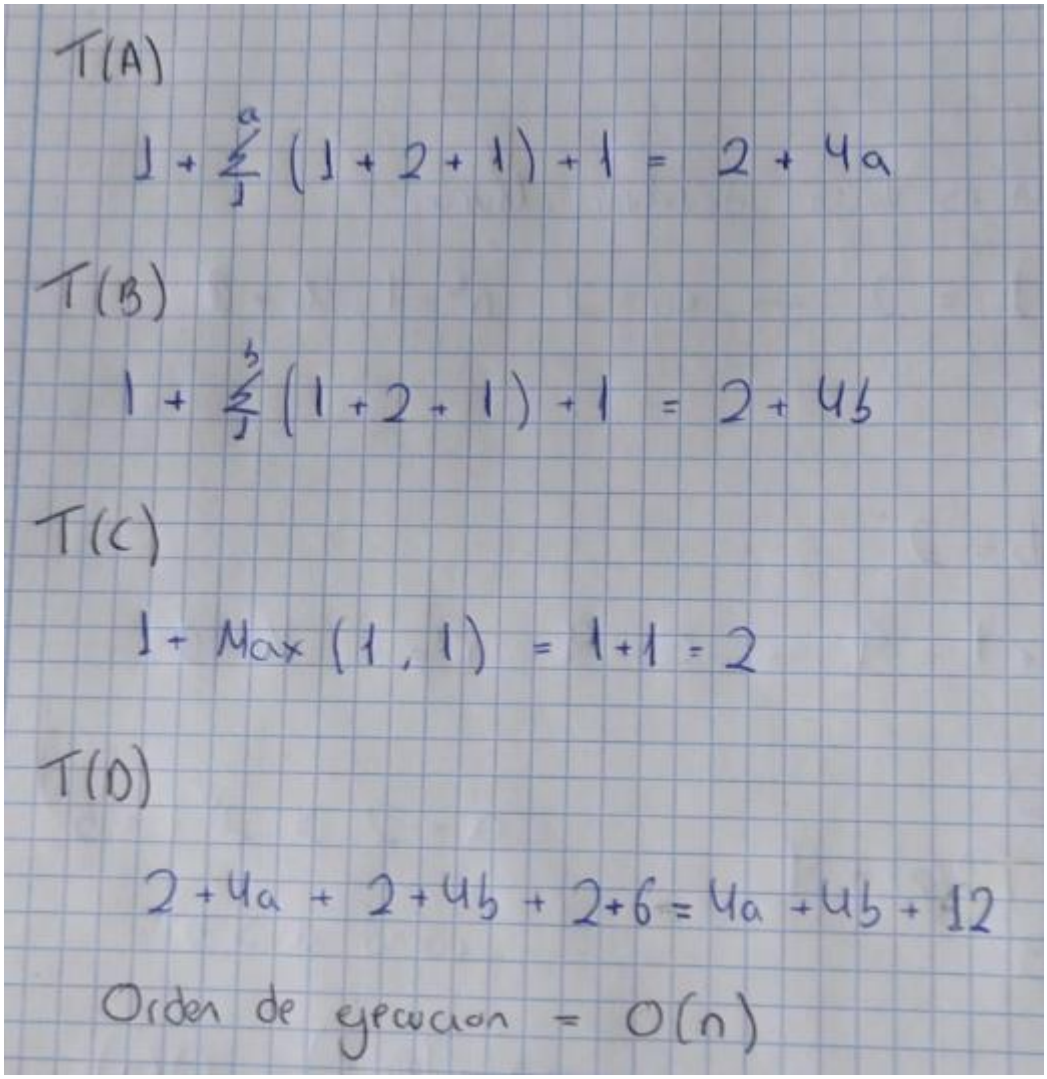
```java
private boolean comprobarVisitados(Lista<Clave> visitados, Clave clave, Clave clave2) {
    int aparece = 0;
    Lista<Clave> a = listaVerticesUnidos(clave);
    Lista<Clave> b = listaVerticesUnidos(clave2);

    for (int i = 1; i <= a.longitud(); i++) {
        if(a.consultar(i) == clave) {
            aparece++;
        }
    }
    for (int i = 1; i <= b.longitud(); i++) {
        if(b.consultar(i) == clave) {
            aparece++;
        }
    }

    if(aparece >= 2) {
        return false;
    }else {
        return true;
    }

}
```

A

D

B

C



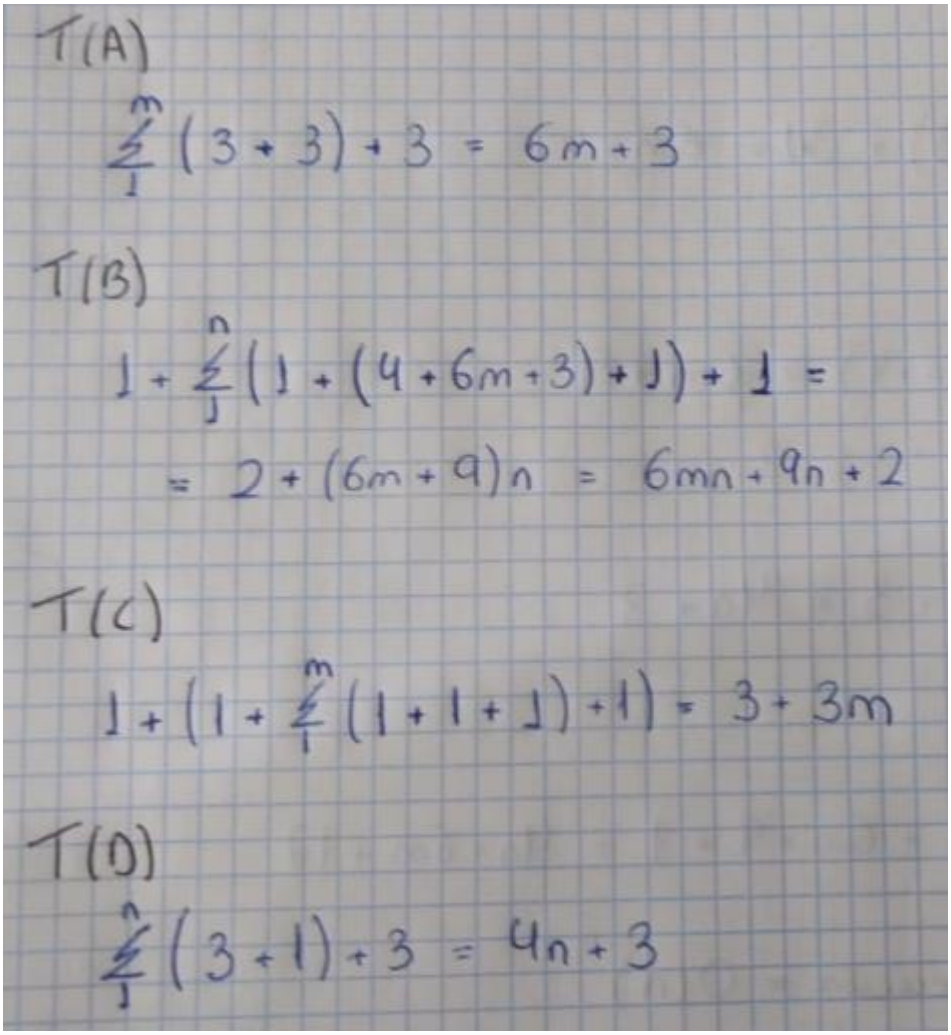
```
private Lista<Clave> listaVerticesUnidos(Clave v) {
    int i = 1;
    Lista<Clave> sucesoresypredecesores = new Lista<Clave>();

    while (i <= vertices.longitud() && !vertices.consultar(i).clave.equals(v)) {
        i++;
    }

    if (i <= vertices.longitud()) {
        for (int j = 1; j <= aristas.consultar(i).longitud(); j++) {
            sucesoresypredecesores.insertar(j,aristas.consultar(i).consultar(j).destino.clave);
        }

    }

    for (int j = 1; j <= vertices.longitud(); j++) {
        int k = 1;
        boolean verticeEncontrado = false;
        while (!verticeEncontrado && k <= aristas.consultar(j).longitud()) {
            if (v.equals(aristas.consultar(j).consultar(k).destino.clave)) {
                sucesoresypredecesores.insertar(1, vertices.consultar(j).clave);
                verticeEncontrado = true;
            } else {
                k++;
            }
        }
    }

    return sucesoresypredecesores;
}
```

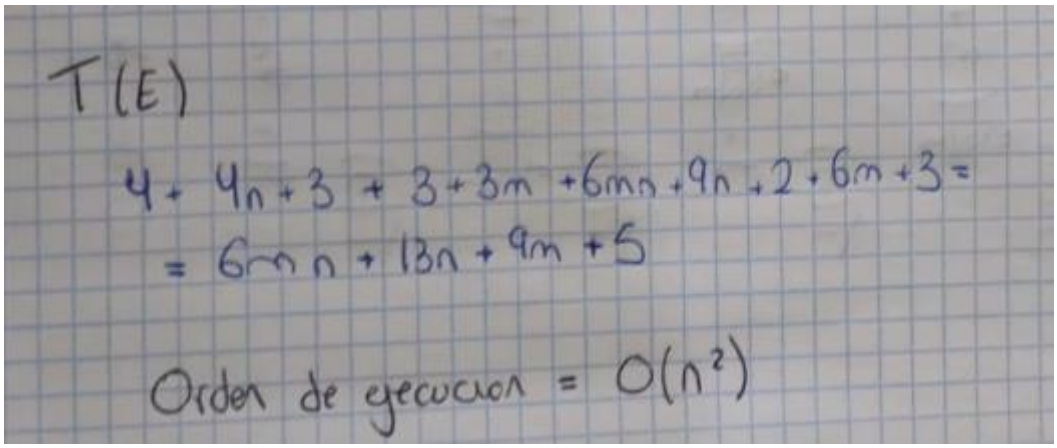
D

C

E

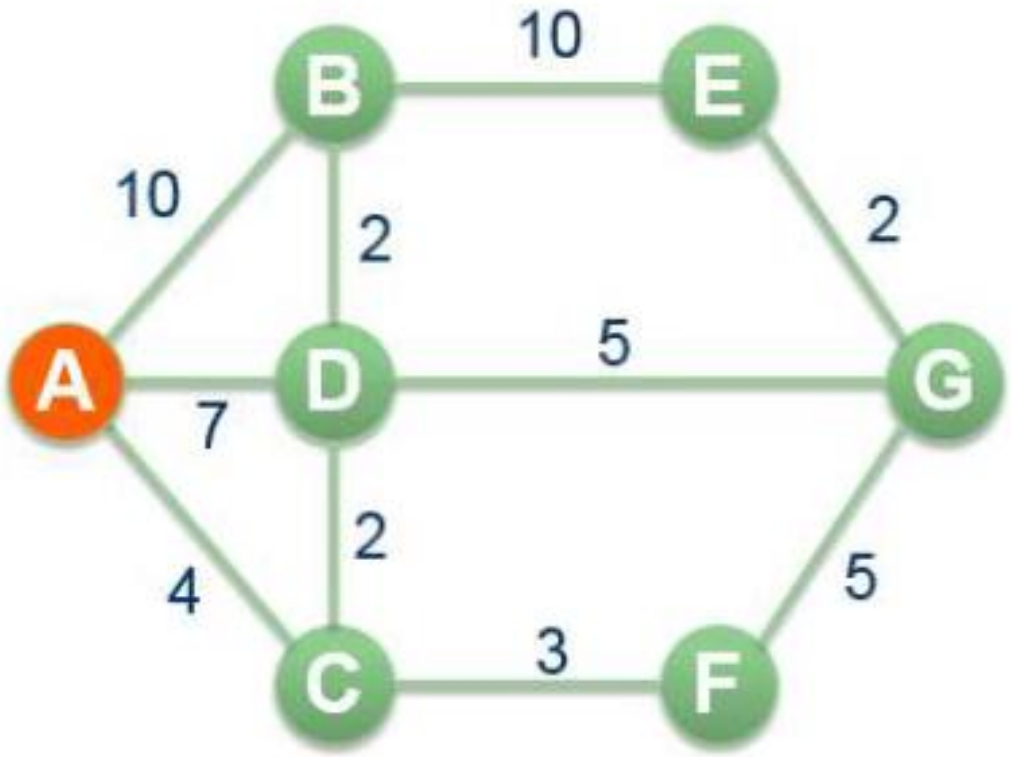
B A




T(E)
4 + 4n + 3 + 3 + 3n + 6mn + 9n + 2 + 6m + 3 =
= 6mn + 13n + 9m + 5
Orden de ejecucion = O(n²)

Resolución ejercicio 3

Aplicar al siguiente grafo el algoritmo de Kruskal. Mostrar el camino a seguir y el coste de las arista de ese camino.



Primero creamos el grafo en java:

```
Nuestro grafo:
G -->
F --> G(5)
E --> G(2)
D --> G(5) C(2) B(2)
C --> F(3)
B --> E(10)
A --> C(4) D(7) B(10)
```

Aplicamos el algoritmo de Kruskal explicado anteriormente:

```
Camino a seguir aplicado el algoritmo de Kriskal:
F --> G
A --> C
C --> F
D --> B
D --> C
E --> G
Coste total de las aristas = 18
```