# Árbol AVL

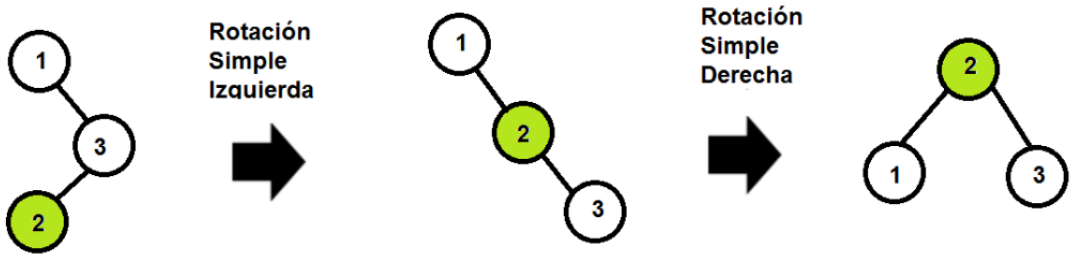
Para la implementación del árbol avl se ha creado una clase nodo previamente. Esta clase posee los atibutos data(que contiene el elemento del nodo), left y right(hijo izquierdo e hijo derecho del propio nodo) y height(altura del mismo).

Para construir la clase nodo se han hecho dos constructores para inicializar los propios atributos del nodo antes descritos.

Ya en la clase del árbol avl,  el único atributo que posee es la raiz, para saber de dónde parte el árbol. Luego, construimos la clase con los métodos básicos, para añadir elementos y hacer las rotaciones que sean necesarias para el equilibrado.

En cuanto al método de añadir un elemento, miramos si la raiz del árbol está vacía, y si es asi, inicializamos el nodo con el elemento a añadir. Si la raiz no es nula y el elemento a añadir es menor que el el elemento que posee la raiz, vamos a hacer recursividad hasta añadirlo en su subarbol iquierdo. Así se hará tambien a la derecha si el elemento pasado por parámetro es mayor que el elemento del nodo.

En las rotaciones, le pasamos un nodo, el cual queremos rotar, ya sea con su hijo derecho o su hijo izquierdo. Aquí un ejemplo de ello:




El árbol también incluye los métodos para recorrelo, inOrder, postOrder y preOrder.

Ejemplo de los recorridos en el programa en funcionamiento:

```
Post order : 4 1 10 7
Pre order : 7 1 4 10
In order : 1 4 7 10
```

# Grafo

Para la implementación de la clase grafo se ha implementado una clase nodo previemente, al igual que en el árbol avl. Ésta clase nodo tiene como atributos un identificador, la distancia y el nodo del que procede (padre).

Como en la anterior clase nodo, en esta también se introducen dos constructores que inicializan los atributos del nodo antes mencionados.

Por último, se han redefinido las clases equals y compareTo para que las comparaciones se hagan mediante los identificadores de los nodos y la distancia.

Empezando en la clase grafo, vemos que tiene un array de cadenas con los identificadores de cada nodo, una matriz de adyacencia, una cadena con la ruta más corta, un entero con esa distancia y una lista con los nodos ya revisados. Esta clase es la que contiene el algoritmo de búsqueda más óptimo entre nodos (algoritmo de Dijkstra).

Este algoritmo realiza estos sencillos pasos:

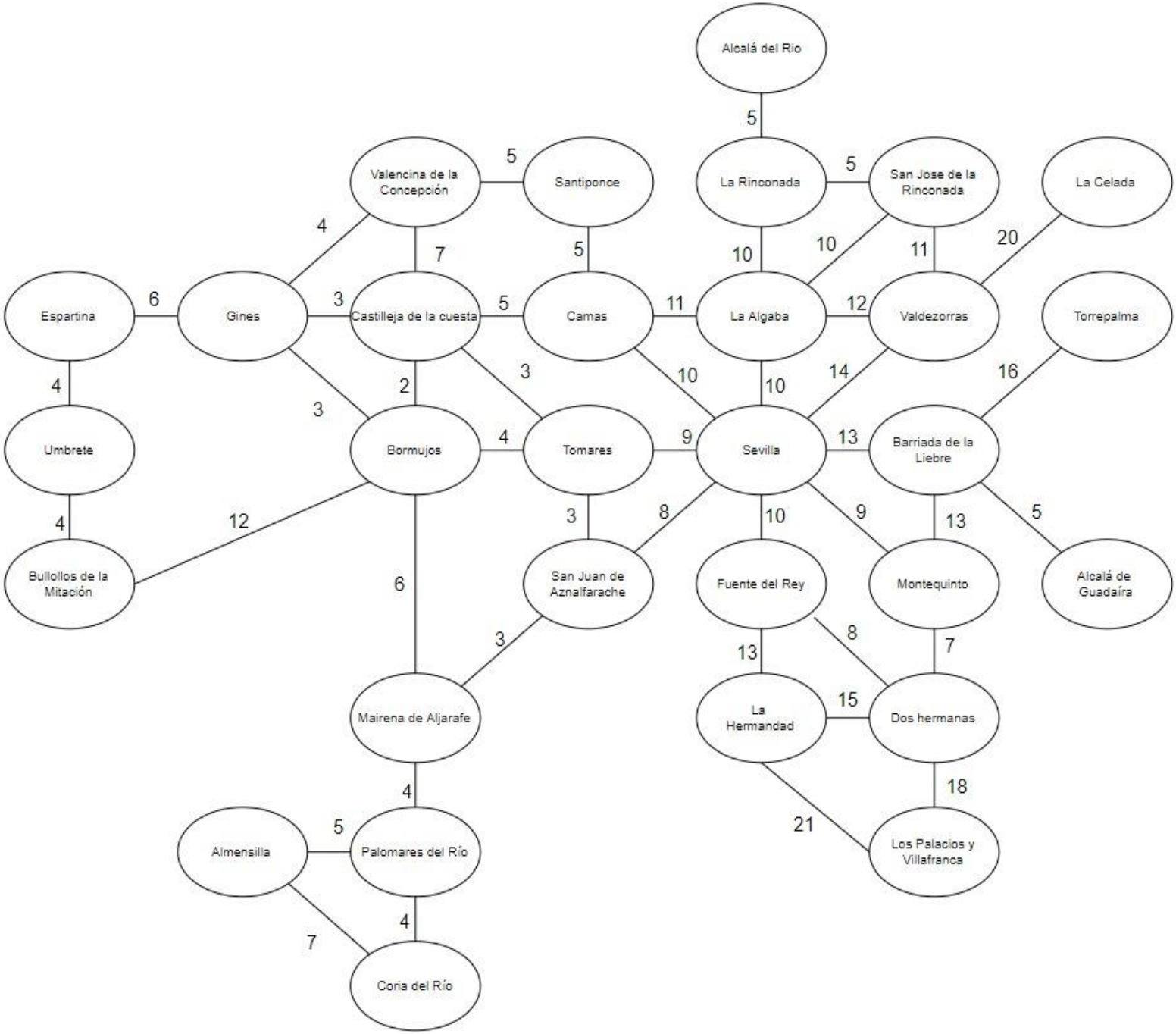
1. Marca el nodo inicial que elegiste con una distancia actual de 0 y el resto con infinito.
2. Establece el nodo no visitado con la menor distancia actual como el nodo actual A.
3. Para cada vecino V de tu nodo actual A: suma la distancia actual de A con el peso de la arista que conecta a A con V. Si el resultado es menor que la distancia actual de V, establécelo como la nueva distancia actual de V.
4. Marca el nodo actual A como visitado.
5. Si hay nodos no visitados, ve al paso 2.

Ejemplo del algoritmo de dijkstra en funcionamiento en el grafo:

```
Introduzca inicio ruta
Sevilla

Introduzca fin ruta
Dos Hermanas
Distancia: 16Km: Sevilla Montequinto Dos Hermanas
```

A continuación, se muetra el mapa con las ciudades y distancias que se ha utilizado para la prueba del grafo y el algoritmo de dijkstra:


Alcalá del Rio
5
Valencina de la Concepción
5
Santiponce
La Rinconada
5
San Jose de la Rinconada
La Celada
4
7
5
10
10
11
20
Espartina
6
Gines
3
Castilleja de la cuesta
5
Camas
11
La Algaba
12
Valdezorras
Torrepalma
4
2
3
10
10
14
16
Umbrete
3
Bormujos
4
Tomares
9
Sevilla
13
Barriada de la Liebre
4
12
3
8
10
9
13
5
Bullollos de la Mitación
6
San Juan de Aznalfarache
Fuente del Rey
Montequinto
Alcalá de Guadaíra
3
13
8
7
Mairena de Aljarafe
La Hermandad
15
Dos hermanas
4
18
5
21
Almensilla
Palomares del Río
Los Palacios y Villafranca
4
7
Coria del Río