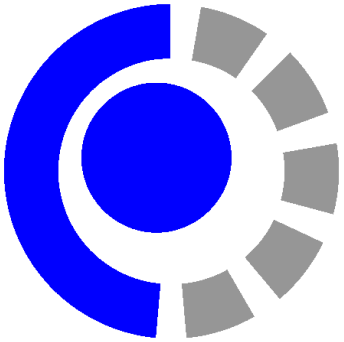
TRABAJO FINAL - Red de Distribución de Agua 2025



Facultad de Informática
UNIVERSIDAD NACIONAL DEL COMAHUE

- ***Estudiantes***
  - *Aguirre, Leandro Nicolás - FAI-5174*
  - *Font Agustín - FAI-5317*
  - *Ramos Montiel Dante - FAI-5314*

- ***Grupo*** *- Grupo n°8*

- ***Materia*** *- Estructuras de Datos*

- ***Profesor/a*** *- Gabriela Aranda*

- ***Fecha De Entrega*** *- 05/08 13:00hs*

- ***Año*** *- 2025*

Árbol de directorios

```
TPFinal-EDAT/
├── bin/
│   ├── clases/
│   │   ├── conjuntistas/
│   │   │   ├── ArbolAVL.class
│   │   │   ├── ArbolBB.class
│   │   │   ├── NodoABB.class
│   │   │   └── NodoAVL.class
│   │   ├── grafos/
│   │   │   ├── Grafo.class
│   │   │   ├── NodoAdy.class
│   │   │   ├── NodoVert.class
│   │   │   └── testGrafo.class
│   │   └── lineales/
│   │       └── dinamicas/
│   │           ├── Cola.class
│   │           ├── Lista.class
│   │           └── Nodo.class
│   ├── main/
│   │   ├── Ciudad.class
│   │   ├── Criterios.class
│   │   ├── DiccionarioAVL.class
│   │   ├── Dom.class
│   │   ├── NodoAVLDicc.class
│   │   └── Tuberia.class
│   └── TransporteDeAgua.class
├── src/
│   ├── clases/
│   │   ├── conjuntistas/
│   │   │   ├── ArbolAVL.java
│   │   │   ├── ArbolBB.java
│   │   │   ├── NodoABB.java
│   │   │   └── NodoAVL.java
│   │   ├── especifico/
│   │   │   ├── DiccionarioAVL.java
│   │   │   ├── main.java
│   │   │   └── NodoAVLDicc.java
│   │   ├── grafos/
```

```
│   │   │   ├── Grafo.java
│   │   │   ├── NodoAdy.java
│   │   │   ├── NodoVert.java
│   │   │   └── testGrafo.java
│   │   └── lineales/
│   │       └── dinamicas/
│   │           ├── Cola.java
│   │           ├── Lista.java
│   │           └── Nodo.java
│   ├── main/
│   │   ├── Ciudad.java
│   │   ├── Criterios.java
│   │   ├── Dom.java
│   │   └── Tuberia.java
│   └── TransporteDeAgua.java
├── textos/
│   ├── datosHabitantes/
│   │   ├── BARILOCHE.txt
│   │   ├── CORDOBA.txt
│   │   ├── MENDOZA.txt
│   │   ├── NEUQUEN.txt
│   │   ├── PARANA.txt
│   │   ├── POSADAS.txt
│   │   ├── RAWSON.txt
│   │   ├── RESISTENCIA.txt
│   │   ├── RIO GALLEGOS.txt
│   │   ├── ROSARIO.txt
│   │   ├── SALTA.txt
│   │   ├── SANTA FE.txt
│   │   ├── TANDIL.txt
│   │   ├── TUCUMAN.txt
│   │   └── USHUAIA.txt
│   ├── ciudades.txt
│   └── tuberias.txt
└── README
```

Clases utilizadas

- **Conjuntistas**
  - ArbolAVL
  - ArbolBB
  - NodoABB
  - NodoAVL

- **Especificos**
  - DiccionarioAVL: implementa un diccionario utilizando un árbol AVL, donde cada nodo almacena una clave y un dato asociado (Nombre de la ciudad - Objeto Ciudad).

  - NodoAVLDicc: Representa un nodo dentro de un árbol AVL utilizado como diccionario. Cada nodo almacena una clave de tipo Comparable y un dato asociado de tipo Object, además de referencias a sus hijos izquierdo y derecho, y su altura dentro del árbol.

- **Grafos**
  - Grafo: representa un grafo dirigido mediante listas enlazadas de vértices y adyacencias. Cada vértice se almacena como un nodo que contiene el elemento asociado y una lista de adyacencias (arcos) hacia otros vértices, cada una con su respectiva etiqueta ().

  - NodoAdy: representa un nodo de adyacencia dentro de la estructura de un grafo. Cada objeto de esta clase almacena una referencia a un vértice destino (NodoVert), una referencia al siguiente nodo de adyacencia en la lista, y una etiqueta asociada al arco (Caudal máximo).

- NodoVert: representa un vértice dentro de la estructura de un grafo. Cada objeto de esta clase almacena el elemento asociado al vértice, una referencia al siguiente vértice en la lista de vértices y una referencia al primer nodo de su lista de adyacencias.

- **Lineales**
  - Cola
  - Lista
  - Nodo

- **Propias del dominio**
  - Ciudad: representa una ciudad y almacena información relevante como su nombre, nomenclatura, superficie, y el promedio de consumo personal de agua por día. Además, mantiene una lista de años, donde cada año contiene un arreglo con la cantidad de habitantes por mes.

  - Dom: representa un par de nombres, generalmente utilizados para identificar una relación entre dos entidades, como ciudades conectadas por una tubería. Almacena dos cadenas de texto (nom1 y nom2) y proporciona métodos para acceder y modificar estos valores.

  - Tuberia: representa una tubería dentro del sistema, almacenando información relevante como su nomenclatura, caudal mínimo y máximo, diámetro y estado.

- TransporteDeAgua: implementa el programa principal para gestionar un sistema de transporte de agua entre ciudades. Se encarga de cargar los datos de ciudades y tuberías desde archivos, almacenando la información en un grafo para las conexiones, un diccionario AVL para las ciudades y un HashMap para las tuberías.

Consulta sobre tuberías de Agua

A continuación se explicarán los métodos y planteamientos para realizar las consultas sobre tuberías de agua en la clase principal TransporteDeAgua:

En consultas sobre transporte de agua, es posible elegir dos opciones.

1. Obtener el camino con caudal pleno mínimo desde A a B.
2. Obtener el camino de A a B pasando por la mínima cantidad de ciudades.

Obtener el camino con caudal pleno mínimo

Primero, se obtienen las nomenclaturas de las ciudades previamente solicitadas, y se verifica si el Grafo no está vacío (no hay ciudades con tuberías).
A continuación, se crea una Lista la cual es el retorno del método obtenerTodosLosCaminos(origen, destino) del Grafo. Este método, como su nombre indica, recorre todos los nodos adyacentes de la ciudad "origen" para verificar si eventualmente se llega a la ciudad "destino", y guarda ese camino en una lista dentro de la Lista de retorno.

El método obtenerTodosLosCaminos es el siguiente

```
public Lista obtenerTodosLosCaminos(Object origen, Object destino) {
    Lista caminos = new Lista();
    NodoVert nodoOrigen = ubicarVertice(origen);
    NodoVert nodoDestino = ubicarVertice(destino);

    if (nodoOrigen != null && nodoDestino != null) {
        Lista caminoActual = new Lista();
        Lista visitados = new Lista(); // reemplazo de HashSet
        obtenerTodosLosCaminosAux(nodoOrigen, destino, caminoActual, caminos, visitados);
    }

    return caminos;
}

private void obtenerTodosLosCaminosAux(NodoVert actual, Object destino, Lista caminoActual, Lista caminos,
        Lista visitados) {
    visitados.insertar(actual.getElem(), visitados.longitud() + 1);
    caminoActual.insertar(actual.getElem(), caminoActual.longitud() + 1);

    if (actual.getElem().equals(destino)) {
        Lista copia = copiarLista(caminoActual);
        caminos.insertar(copia, caminos.longitud() + 1);
    } else {
        NodoAdy ady = actual.getPrimerAdy();
        while (ady != null) {
            NodoVert vecino = ady.getVertice();
            if (!contiene(visitados, vecino.getElem())) {
                obtenerTodosLosCaminosAux(vecino, destino, caminoActual, caminos, visitados);
            }
            ady = ady.getSigAdyacente();
        }
    }

    visitados.eliminar(visitados.longitud());
    caminoActual.eliminar(caminoActual.longitud());
}
```

Volviendo al método de consulta de caudal pleno mínimo, si la lista que retorna el método no está vacía (hay caminos posibles), se procede a buscar el caudal pleno mínimo en cada camino y compararlos.
Para ello, en un método aparte, se realiza un primer "for" para recorrer cada Lista con caminos, y luego otro "for" que recorre la propia Lista del camino en específico. En cada camino, se realiza una comprobación de si el diámetro es el mínimo del camino, y en caso positivo, se guarda ese diámetro y al mismo tiempo el caudal máximo del camino.
Una vez se termina de recorrer todo el camino, se compara ese caudal máximo con el caudal pleno mínimo almacenado (que en la primera iteración siempre se asigna), y en caso de ser mínimo, se sobreescribe.

```java
// Busca el camino con el menor caudal pleno entre los posibles
public static Lista encontrarCaudalPlenoMin(Lista caminosPosibles, HashMap<Dom, Tuberia> hashX) {
    Lista out = new Lista();
    if (!caminosPosibles.esVacia()) {
        int cadaLista = 1;
        Lista aux = (Lista) caminosPosibles.recuperar(1);
        int longi = caminosPosibles.longitud();
        Dom dominio = new Dom((String) aux.recuperar(1), (String) aux.recuperar(2));
        int caudalPlenoMin = 0;
        int caudalPlenoActual = 0;
        int diametroMin = hashX.get(dominio).getDiametro();
        for (int i = 1; i <= longi; i++) {
            aux = (Lista) caminosPosibles.recuperar(i);
            if (aux != null) {
                dominio.setNom1((String) aux.recuperar(1));
                dominio.setNom2((String) aux.recuperar(2));
                diametroMin = hashX.get(dominio).getDiametro();
                caudalPlenoActual = 0;
                for (int g = 1; g < aux.longitud(); g++) {
                    dominio.setNom1((String) aux.recuperar(g));
                    dominio.setNom2((String) aux.recuperar(g + 1));
                    if (hashX.get(dominio).getDiametro() <= diametroMin && !out.equals(aux)) {
                        diametroMin = hashX.get(dominio).getDiametro();
                        caudalPlenoActual = hashX.get(dominio).getCaudalMax();
                    }
                }
                if (caudalPlenoActual < caudalPlenoMin || i == 1) {
                    caudalPlenoMin = caudalPlenoActual;
                    out = aux;
                }
            }
        }
    }
    return out;
}
```

Una vez obtenido el camino con el caudal pleno mínimo, éste se muestra en la terminal.
A continuación, se muestra el método completo de consultarCaminoCaudalPlenoMin:

```java
// Consulta el camino con caudal pleno mínimo entre dos ciudades
public static void consultarCaminoCaudalPlenoMin(Grafo mapa, Ciudad ciudadA, Ciudad ciudadB,
        HashMap<Dom, Tuberia> hashX) {
    String nomenclaturaA = ciudadA.getNomenclatura();
    String nomenclaturaB = ciudadB.getNomenclatura();
    if (!mapa.vacio()) {
        // Obtiene todos los caminos posibles entre las dos ciudades
        Lista caminosPosibles = mapa.obtenerTodosLosCaminos(nomenclaturaA, nomenclaturaB);
        if (!caminosPosibles.esVacia()) {
            // Busca el camino con el menor caudal pleno
            Lista caminoConCaudalMin = encontrarCaudalPlenoMin(caminosPosibles, hashX);
            if (!caminoConCaudalMin.esVacia()) {
                // Muestra el camino y su estado
                System.out.println("El camino con el caudal pleno minimo desde " + ciudadA.getNombre() + " hasta "
                        + ciudadB.getNombre() + " es:");
                System.out.println(caminoConCaudalMin.toString());
                System.out.println("Y el estado del camino actualmente es "
                        + decidirEstadoCamino(obtenerListaEstadosTuberias(caminoConCaudalMin, hashX)));

            } else {
                System.out.println(x:"ERROR: Hubo un error al encontrar la tuberia mas pequeña, saliendo...");
            }
        } else {
            System.out.println("ERROR: No existe ningun camino desde " + ciudadA.getNombre() + " hasta "
                    + ciudadB.getNombre() + ", saliendo...");
        }
    }
}
```

Obtener el camino pasando por la menor cantidad de ciudades

Se obtienen las nomenclaturas, y luego se busca el camino más corto llamando al Grafo con el método caminoMasCorto(origen, destino), que genera una lista y va recorriendo y almacenando el camino hasta destino o hasta que la longitud del camino actual sea mayor al camino más corto almacenado (no es necesario recorrerlo más porque se sabe que es más largo).
A continuación se muestra el método caminoMasCorto de Grafo.

```java
public Lista caminoMasCorto(Object origen, Object destino) {
    Lista camino = new Lista();
    NodoVert vertOrigen = ubicarVertice(origen);
    NodoVert vertDestino = ubicarVertice(destino);
    if (vertOrigen != null && vertDestino != null) {
        Lista visitados = new Lista();
        Lista mejorCamino = new Lista();
        caminoMasCortoAux(vertOrigen, destino, visitados, mejorCamino);
        return mejorCamino;
    }
    return camino;
}

private void caminoMasCortoAux(NodoVert actual, Object destino, Lista visitados, Lista mejorCamino) {
    visitados.insertar(actual.getElem(), visitados.longitud() + 1);

    //Si el nodo actual es el destino
    if (actual.getElem().equals(destino)) {
        //Verificamos si el camino encontrado es el mejor
        if (mejorCamino.longitud() == 0 || visitados.longitud() < mejorCamino.longitud()) {
            //Si lo es, copiamos la lista de visitados al mejor camino
            copiarLista(visitados, mejorCamino);
        }
    } else if(mejorCamino.longitud() == 0 || visitados.longitud() < mejorCamino.longitud()){ //Si no es el destino, seguimos buscando (siempre y cuando sea útil)
        NodoAdy ady = actual.getPrimerAdy();
        while (ady != null) {
            if (!pertenece(visitados, ady.getVertice().getElem())) {
                caminoMasCortoAux(ady.getVertice(), destino, visitados, mejorCamino);
            }
            ady = ady.getSigAdyacente();
        }
    }

    visitados.eliminar(visitados.longitud());
}
```

Una vez se obtiene el camino más corto (si es que existe), se comprueba el estado del camino completo, para ello, se recorre en un método aparte cada camino y se decide el estado del mismo según el estado de sus tuberías, el criterio es el siguiente:

- Una tubería en reparación indica que todo el camino está en reparación.
- Una tubería en diseño indica que todo el camino está en diseño.
- Una tubería inactiva indica que todo el camino está inactivo.
- Si hay una tubería en diseño y otra en reparación o inactiva, el camino se considera en diseño.
- Si hay una tubería en reparación y otra inactiva, el camino se considera inactivo.

Se muestra a continuación los métodos utilizados para decidir el estado del camino:

```java
// Obtiene una lista con los estados de las tuberías de un camino
public static Lista obtenerListaEstadosTuberias(Lista tuberias, HashMap<Dom, Tuberia> hashX) {
    Tuberia aux = null;
    Lista estadoTuberias = new Lista();
    int i = 1;
    Dom dominio = new Dom();
    while ((i + 1) <= tuberias.longitud()) {
        dominio.setNom1((String) tuberias.recuperar(i));
        dominio.setNom2((String) tuberias.recuperar(i + 1));
        aux = hashX.get(dominio);
        if (aux != null) {
            estadoTuberias.insertar(aux.getEstado(), i);
        }
        i++;
    }
    return estadoTuberias;
}
```

```java
// Decide el estado general de un camino según los estados de sus tuberías
public static String decidirEstadoCamino(Lista estadoTuberias) {
    int i = 1;
    int condicion = 0;
    String estadoActual = "";
    while (i <= estadoTuberias.longitud()) {
        estadoActual = (String) estadoTuberias.recuperar(i);
        switch (estadoActual) {
            case "ACTIVO":
                if (condicion == 0) {
                    condicion = 1;
                }
                break;
            case "EN REPARACION":
                if (condicion != 3 || condicion != 4) {
                    condicion = 2;
                }
                break;
            case "EN DISEÑO":
                if (condicion != 4) {
                    condicion = 3;
                }
                break;
            case "INACTIVO":
                condicion = 4;
                break;
            default:
                System.out.println(x:"ERROR");
                condicion = 0;
                break;
        }
        i++;
    }
```

```
    // Devuelve el estado final según la prioridad encontrada
    switch (condicion) {
        case 1:
            estadoActual = "ACTIVO";
            break;
        case 2:
            estadoActual = "EN REPARACION";
            break;
        case 3:
            estadoActual = "EN DISEÑO";
            break;
        case 4:
            estadoActual = "INACTIVO";
            break;
        default:
            estadoActual = "ERROR";
            break;
    }
    return estadoActual;
}
```

Se van decidiendo los estados con números de estado, es decir, si todas las tuberías del camino tienen estado “ACTIVO”, entonces el número de estado del camino es 1, si hay algunas tuberías con otros estados, ya no será posible volver al estado “ACTIVO”, ya que éste se asigna 1 sola vez y debe ser al inicio de la tubería, sino se asume que ese camino no está “ACTIVO”.
Luego, si hay una tubería “EN REPARACIÓN” y ninguna otra tubería está “EN DISEÑO” o “INACTIVO”, entonces se asigna el número de estado 2, en caso contrario, no se podrá volver al estado “EN REPARACIÓN”. Después, si hay una tubería “EN DISEÑO” y no hay ninguna “INACTIVO”, entonces el camino tiene el número de estado 3. Y por último, si hay una tubería en “INACTIVO”, se asigna al camino el número de estado 4.
Por último, en el método principal, se muestra tanto el camino como el estado del mismo. El código realizado fue el siguiente:

```java
// Consulta el camino más corto (menos ciudades) entre dos ciudades
public static void consultarCaminoMinimo(Grafo mapa, Ciudad ciudadA, Ciudad ciudadB, HashMap<Dom, Tuberia> hashX) {
    String nomenclaturaA = ciudadA.getNomenclatura();
    String nomenclaturaB = ciudadB.getNomenclatura();
    // Obtiene el camino más corto
    Lista caminoMinimo = mapa.caminoMasCorto(nomenclaturaA, nomenclaturaB);
    if (!caminoMinimo.esVacia()) {
        // Obtiene y muestra el estado del camino
        Lista estadoTuberias = obtenerListaEstadosTuberias(caminoMinimo, hashX);
        String estadoCamino = decidirEstadoCamino(estadoTuberias);
        if (!estadoCamino.equals("ERROR")) {
            System.out.println("El camino mas corto desde " + ciudadA.getNombre() + " hasta " + ciudadB.getNombre()
                    + " es el siguiente:");
            System.out.println(caminoMinimo.toString());
            System.out.println("Y el estado del camino completo es: " + estadoCamino);
        } else {
            System.out.println("ERROR: Volviendo al menu principal...");
        }
    } else {
        System.out.println(
                "ERROR: No existe un camino desde " + ciudadA.getNombre() + " hasta " + ciudadB.getNombre());
    }
}
```

Estructuras

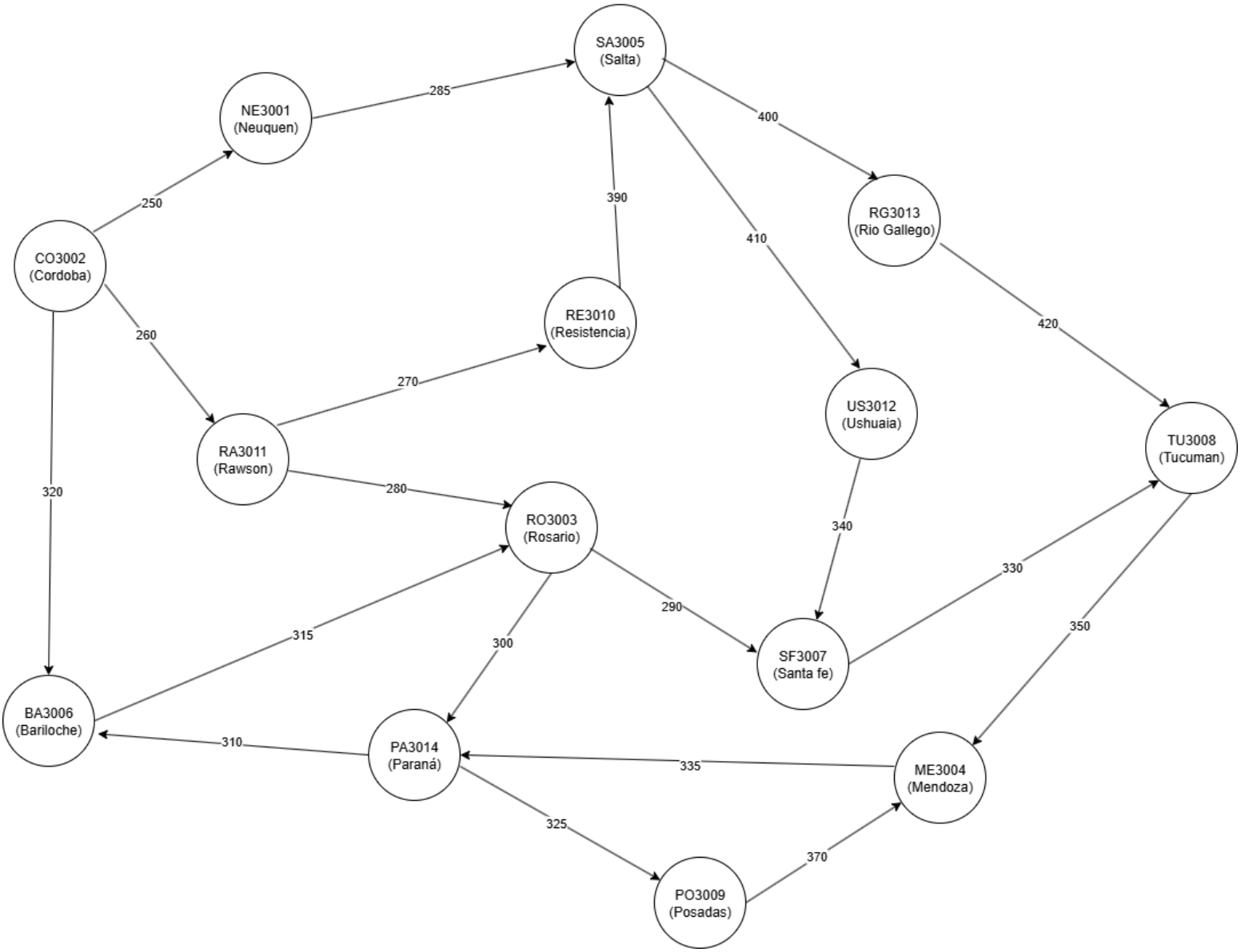
- ***Mapa (Grafo):*** se utiliza para modelar la red de conexiones entre las ciudades mediante tuberías. Cada vértice del grafo representa una ciudad (identificada por su nomenclatura), y cada arco representa una tubería que conecta dos ciudades, almacenando información relevante como el caudal máximo.

- ***nomenclaturasLibres (Cola):*** Tiene el objetivo de ser un contador auxiliar para las nomenclaturas de manera que almacena el valor más grande registrado. Es utilizado para crear nuevas ciudades.

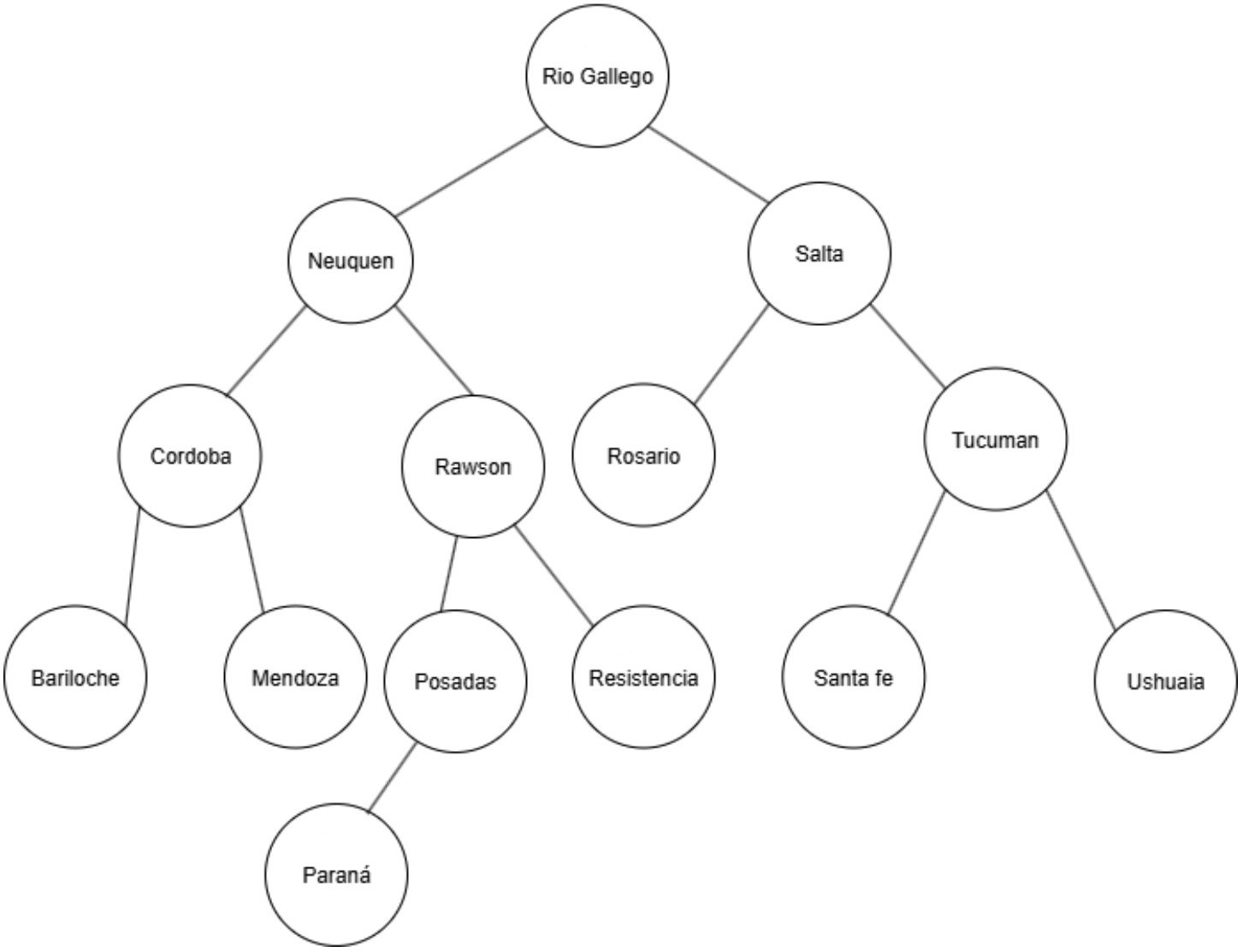
- ***hashTuberias (HashMap nativo):*** se utiliza para almacenar y gestionar todas las tuberías que conectan las ciudades. La **clave** es un objeto Dom, que representa el par de nomenclaturas de las ciudades de origen y destino de la tubería, y el **valor** es un objeto Tuberia que contiene toda la información relevante de esa tubería.

- ***arbolCiudades (DiccionarioAVL):*** se utiliza para almacenar y gestionar todas las ciudades del dominio. La **clave** es el nombre de la ciudad (en mayúsculas) y el **dato asociado** es un objeto de la clase Ciudad, que contiene toda la información relevante de esa ciudad. Los objetos Ciudad se guardan en un arbol AVL ordenados por su clave (nombre).

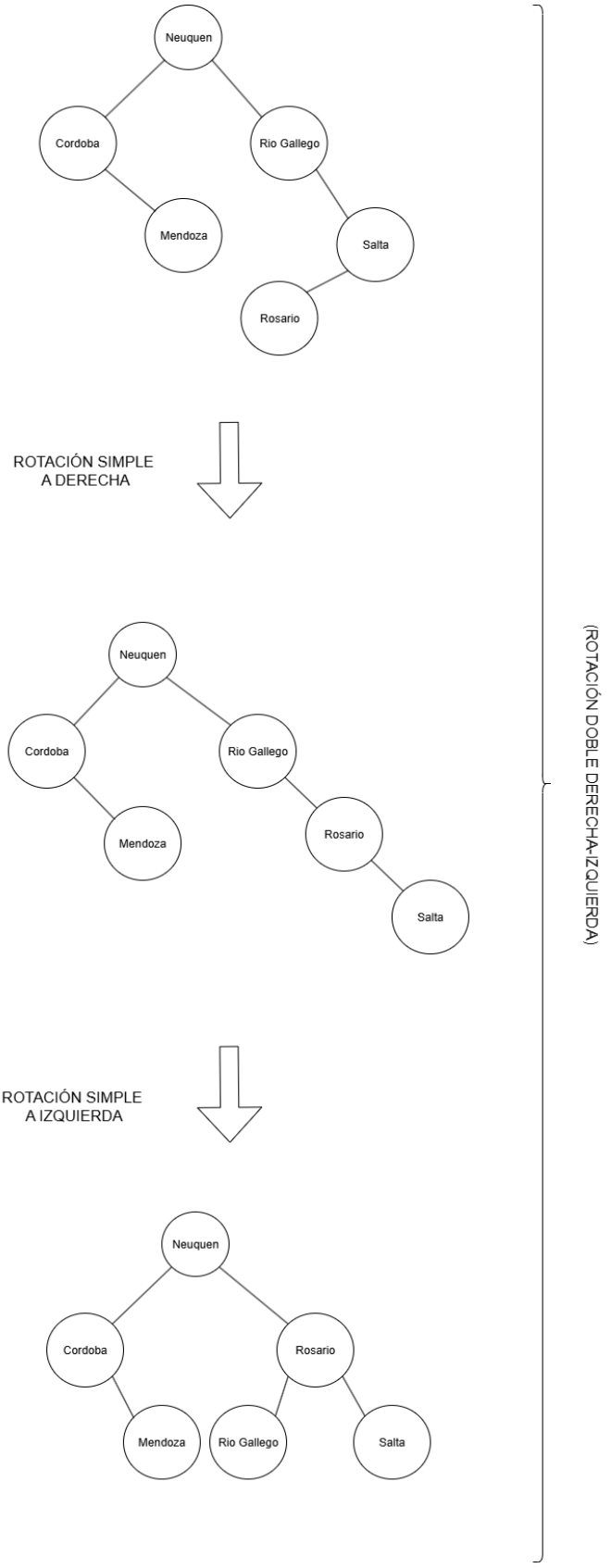
GRAFO

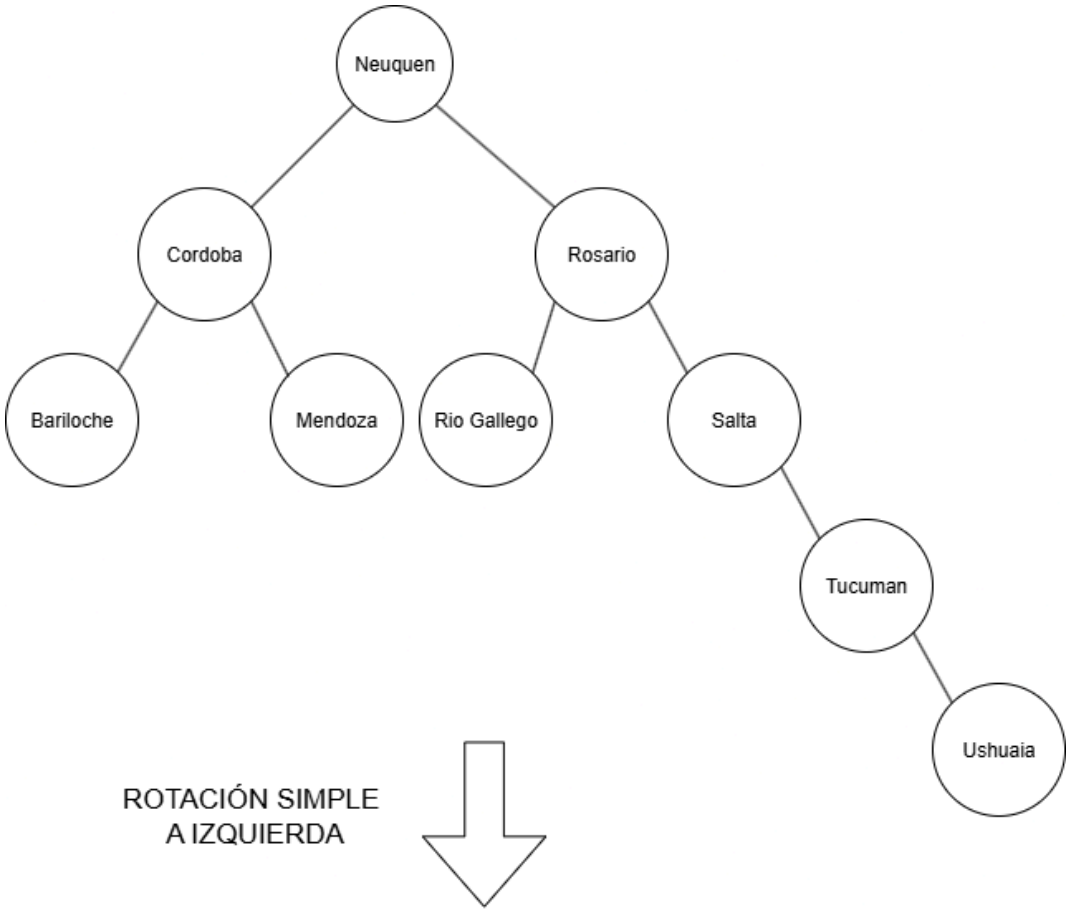

SA3005
(Salta)
NE3001
(Neuquen)
285
400
RG3013
(Rio Gallego)
250
390
410
CO3002
(Cordoba)
RE3010
(Resistencia)
420
260
270
US3012
(Ushuaia)
TU3008
(Tucuman)
RA3011
(Rawson)
320
280
RO3003
(Rosario)
340
330
290
315
300
350
SF3007
(Santa fe)
BA3006
(Bariloche)
310
PA3014
(Paraná)
335
ME3004
(Mendoza)
325
370
PO3009
(Posadas)

ARBOL FINAL

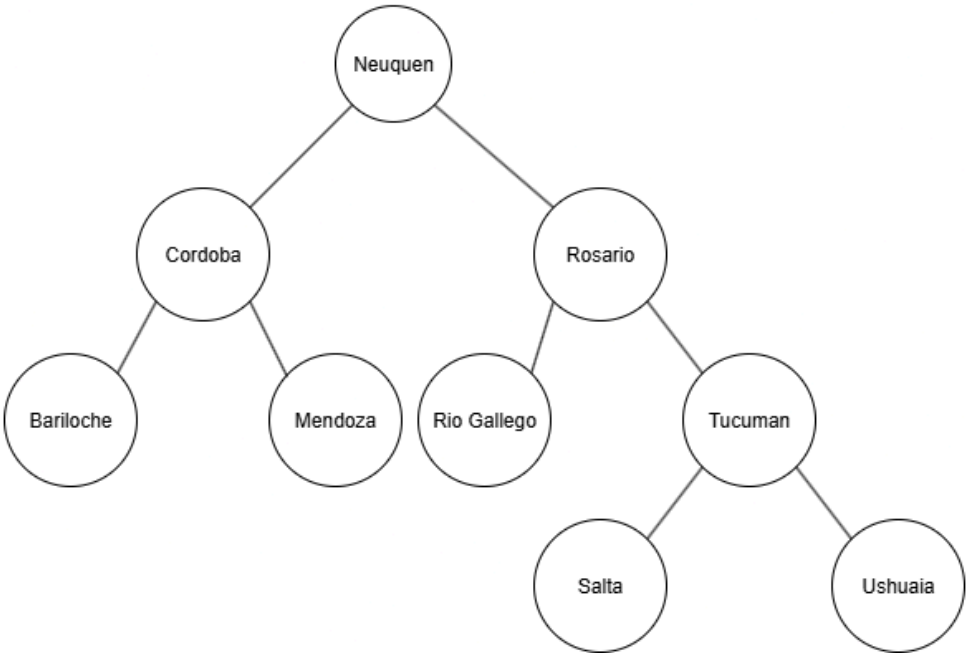

Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Posadas
Resistencia
Santa fe
Ushuaia
Paraná

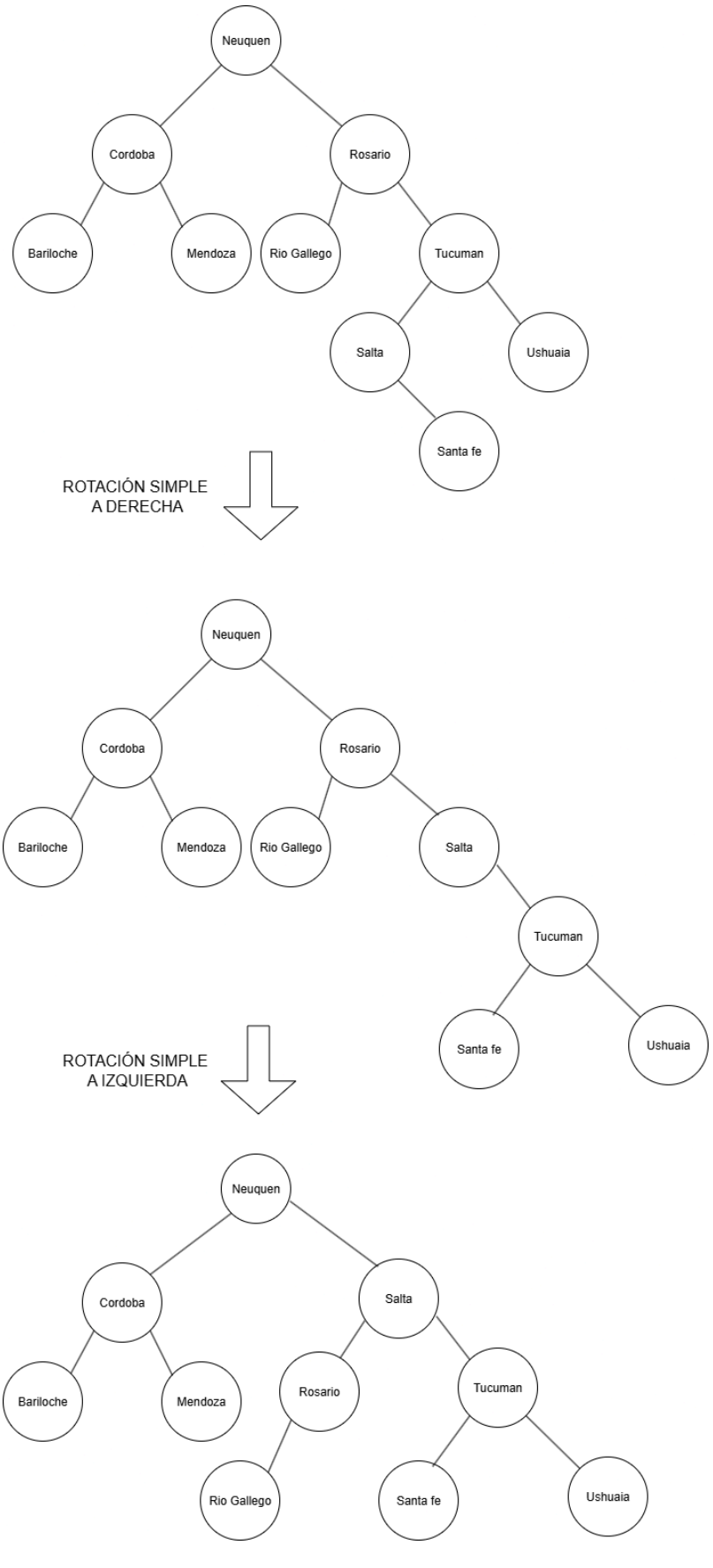
Evolución del Árbol AVL

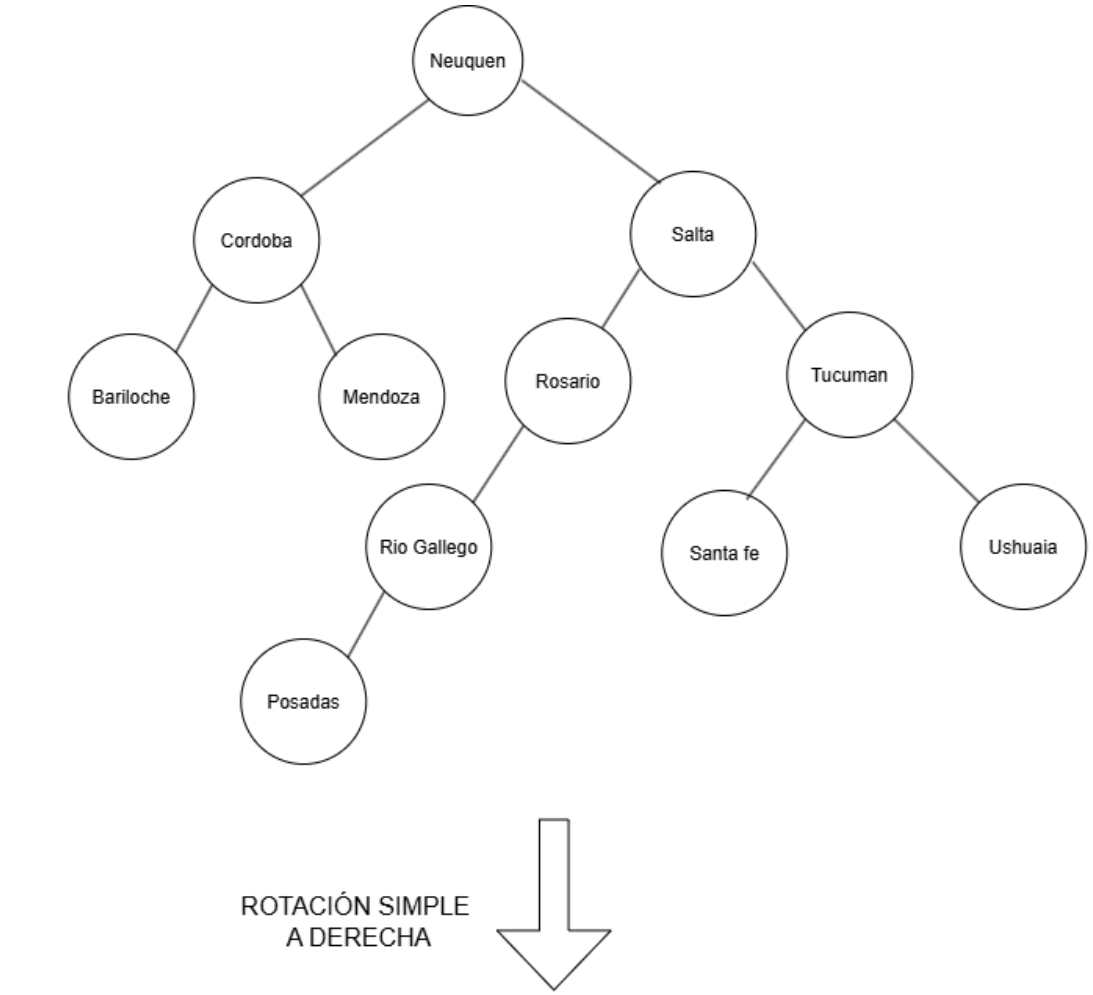




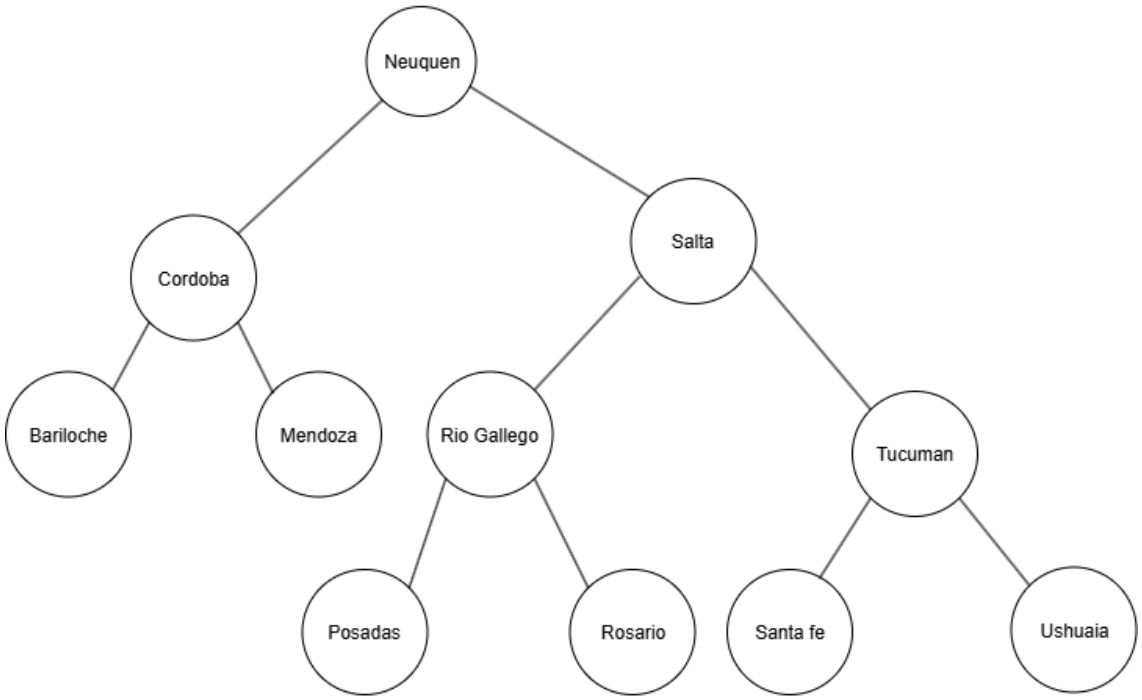
ROTACIÓN SIMPLE
A IZQUIERDA

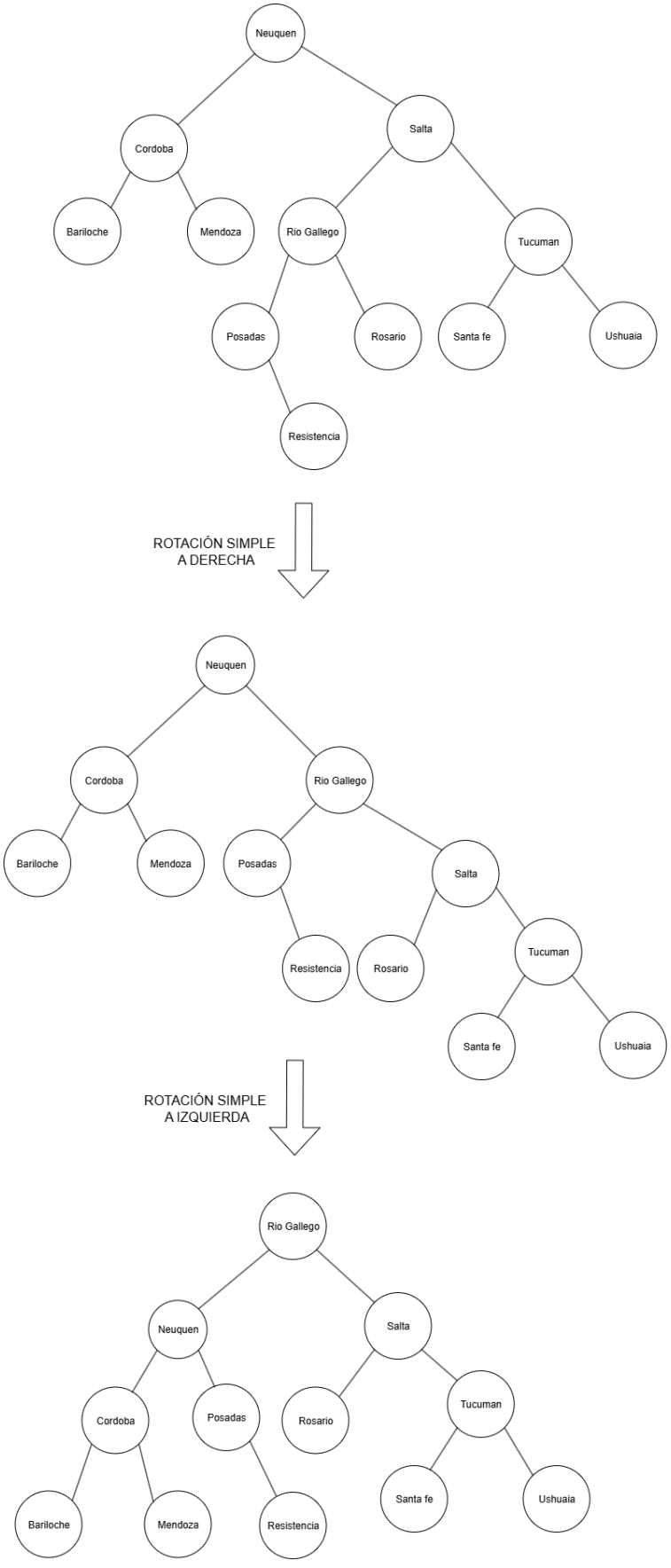


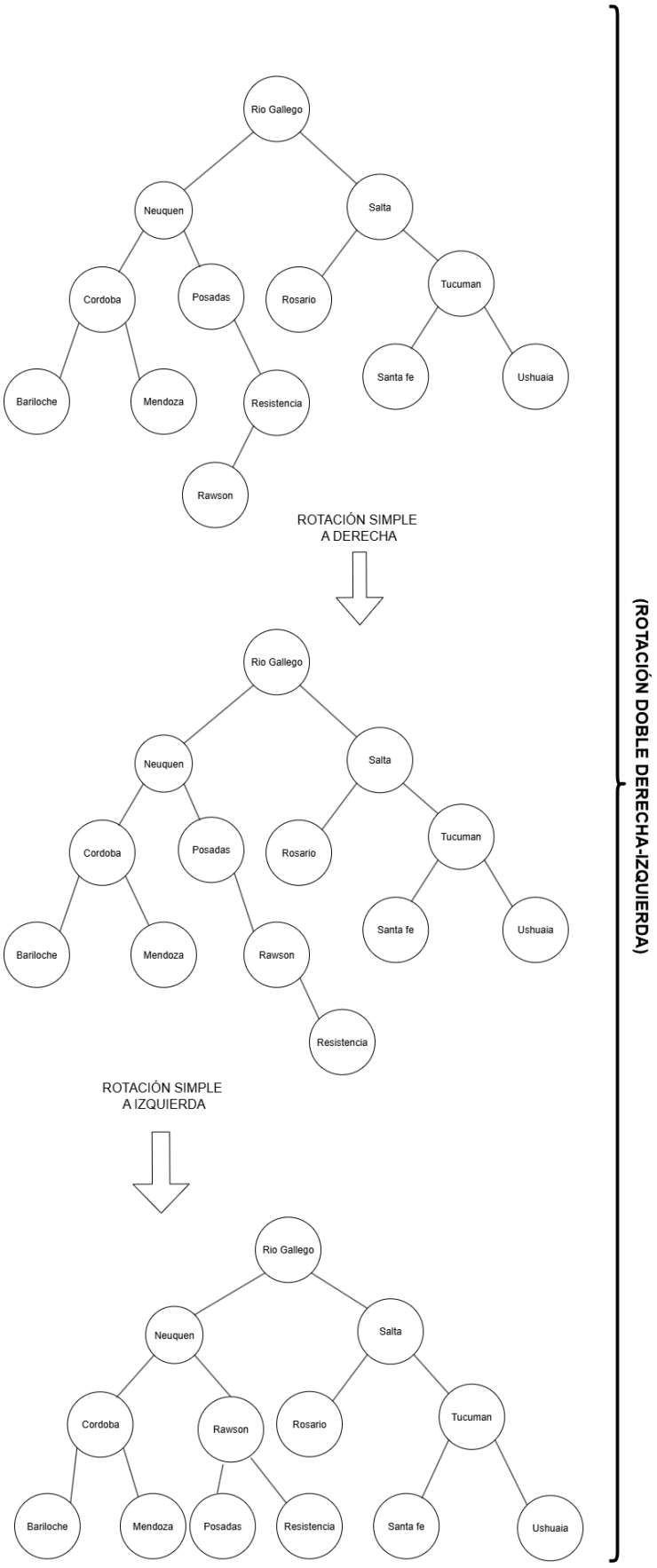

Neuquen
Cordoba
Rosario
Bariloche
Mendoza
Rio Gallego
Tucuman
Salta
Ushuaia
Santa fe
ROTACIÓN SIMPLE
A DERECHA
Neuquen
Cordoba
Rosario
Bariloche
Mendoza
Rio Gallego
Salta
Tucuman
Santa fe
Ushuaia
ROTACIÓN SIMPLE
A IZQUIERDA
Neuquen
Cordoba
Salta
Bariloche
Mendoza
Rosario
Tucuman
Rio Gallego
Santa fe
Ushuaia



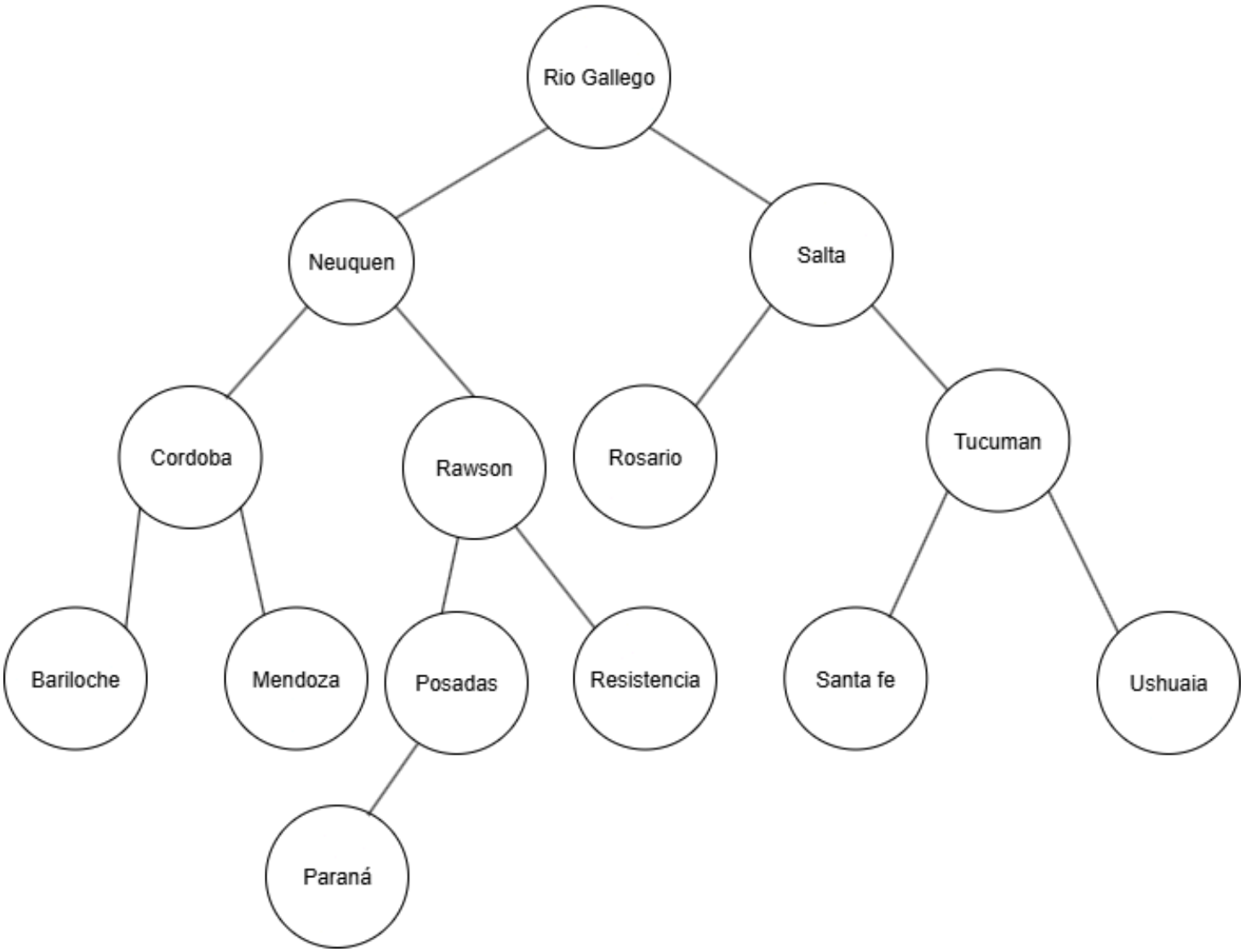
ROTACIÓN SIMPLE
A DERECHA



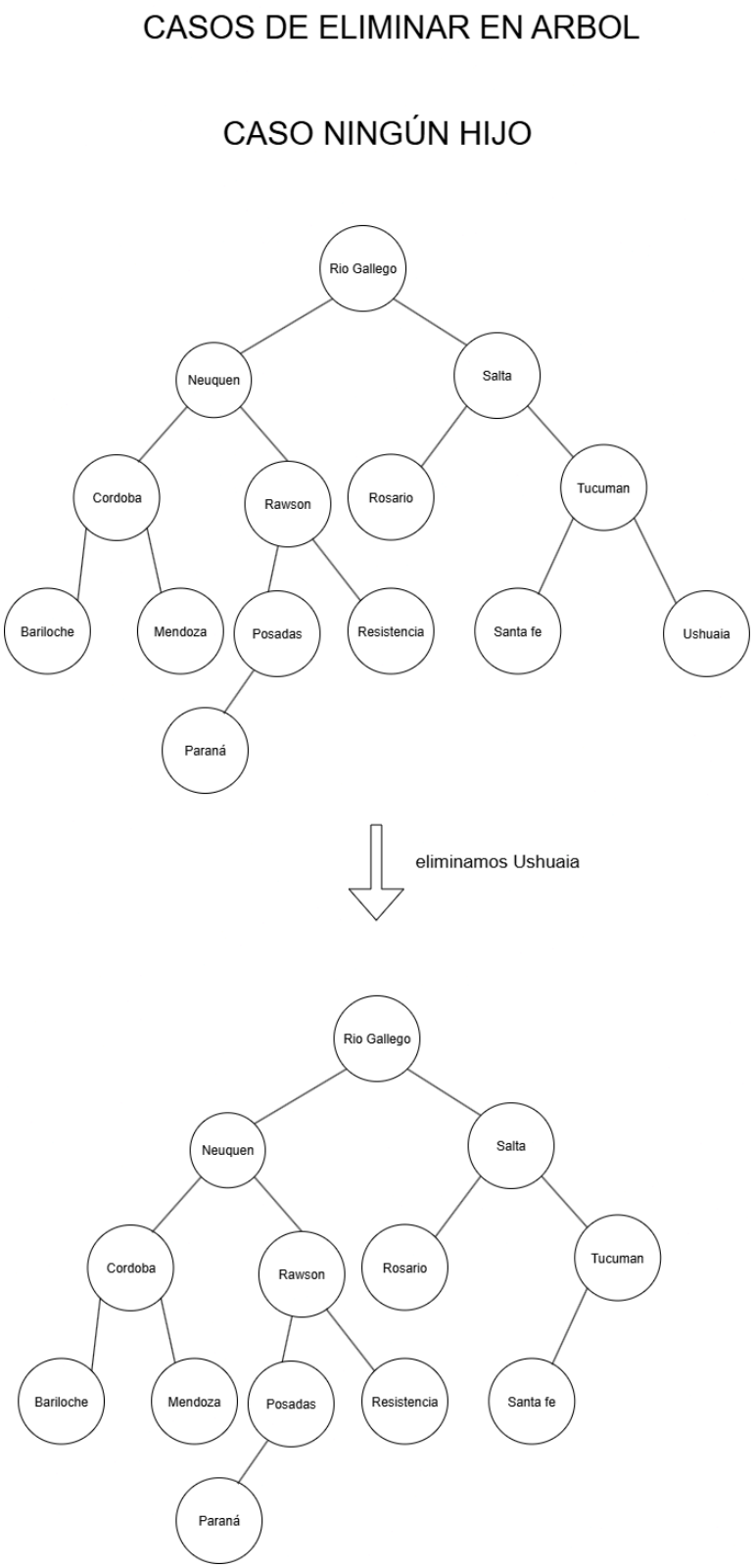

Neuquen
Cordoba
Salta
Bariloche
Mendoza
Rio Gallego
Tucuman
Posadas
Rosario
Santa fe
Ushuaia
Resistencia
ROTACIÓN SIMPLE
A DERECHA
Neuquen
Cordoba
Rio Gallego
Bariloche
Mendoza
Posadas
Salta
Resistencia
Rosario
Tucuman
Santa fe
Ushuaia
ROTACIÓN SIMPLE
A IZQUIERDA
Rio Gallego
Neuquen
Salta
Cordoba
Posadas
Rosario
Tucuman
Santa fe
Ushuaia
Bariloche
Mendoza
Resistencia


Rio Gallego
Neuquen
Salta
Cordoba
Posadas
Rosario
Tucuman
Bariloche
Mendoza
Resistencia
Santa fe
Ushuaia
Rawson
ROTACIÓN SIMPLE
A DERECHA
Rio Gallego
Neuquen
Salta
Cordoba
Posadas
Rosario
Tucuman
Bariloche
Mendoza
Rawson
Santa fe
Ushuaia
Resistencia
ROTACIÓN SIMPLE
A IZQUIERDA
Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Posadas
Resistencia
Santa fe
Ushuaia
(ROTACIÓN DOBLE DERECHA-IZQUIERDA)

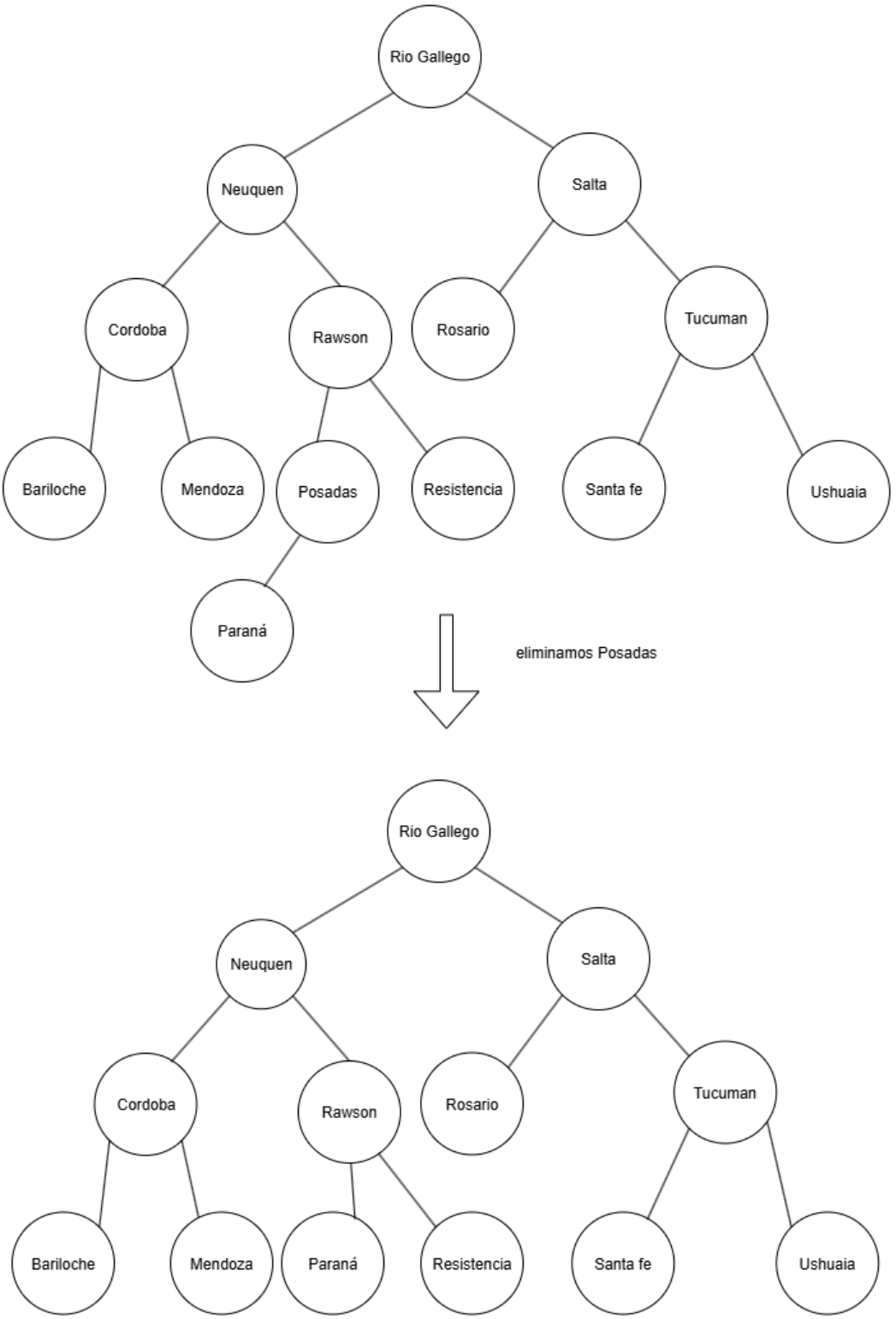
ARBOL FINAL


Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Posadas
Resistencia
Santa fe
Ushuaia
Paraná

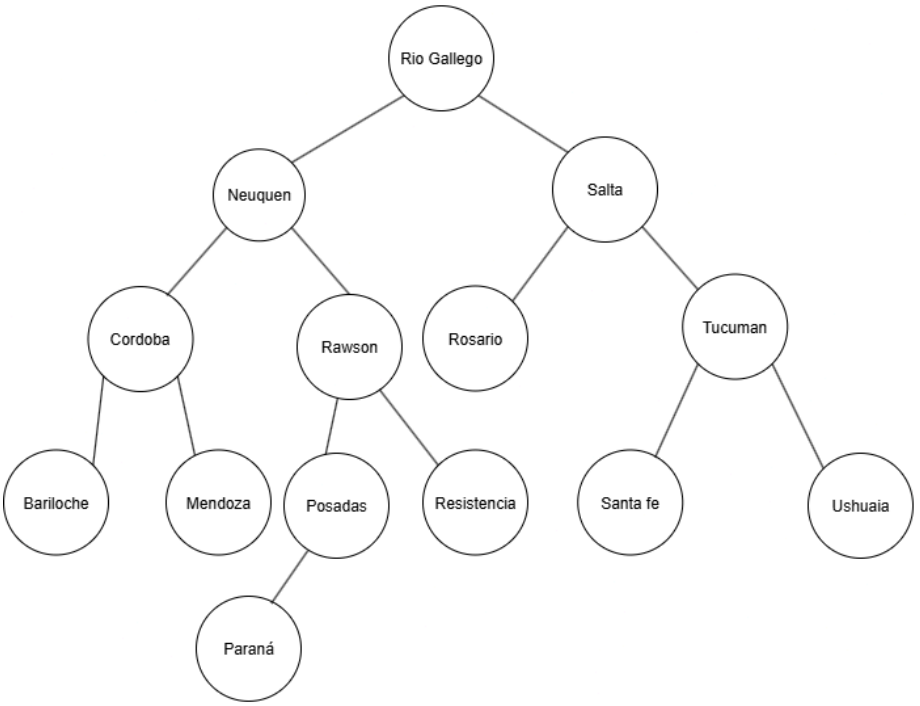
Casos Especiales del Árbol AVL



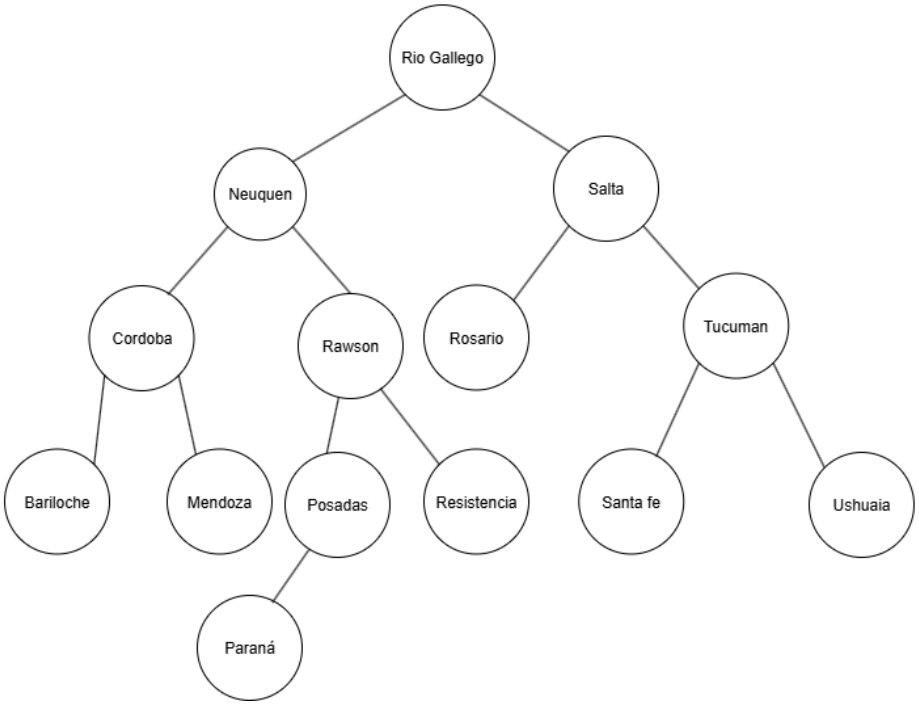
CASO UN HIJO

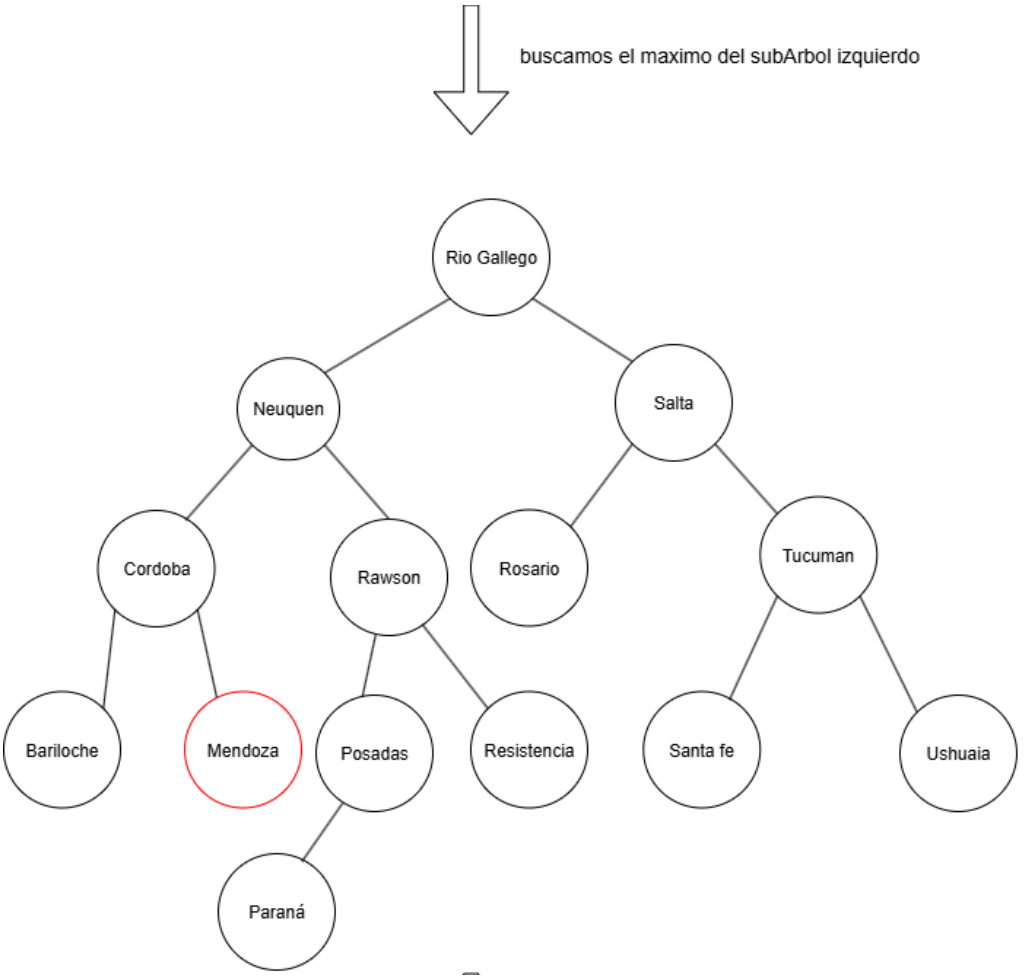

Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Posadas
Resistencia
Santa fe
Ushuaia
Paraná
eliminamos Posadas
Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Paraná
Resistencia
Santa fe
Ushuaia

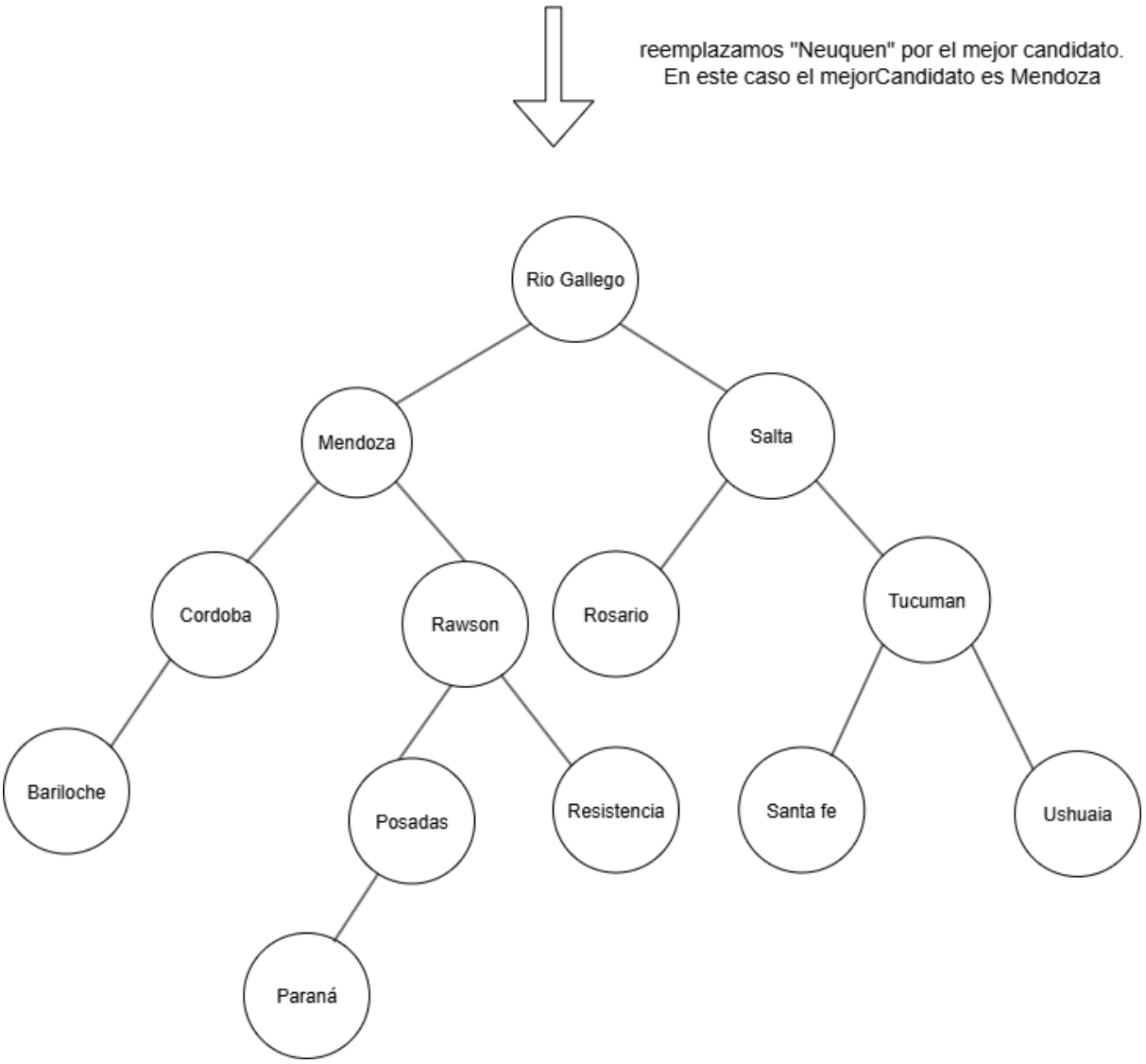
CASO DOS HIJOS



primero se busca el mejor candidato
clave a eliminar "Neuquen"




buscamos el maximo del subArbol izquierdo
Rio Gallego
Neuquen
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Mendoza
Posadas
Resistencia
Santa fe
Ushuaia
Paraná


reemplazamos "Neuquen" por el mejor candidato.
En este caso el mejorCandidato es Mendoza
Rio Gallego
Mendoza
Salta
Cordoba
Rawson
Rosario
Tucuman
Bariloche
Posadas
Resistencia
Santa fe
Ushuaia
Paraná