Manual Técnico

Proyecto: GPS
Lenguaje: Java
ID: NetBeans 8.2

Para la creación de este proyecto se tuvieron que implementar dos principales estructuras de Datos los cuales son:

- Árbol
- Grafos Direccionales

En el siguiente manual se le explicara las funciones y estructuras de cada una de las anteriores.

Grafos:
En el proyecto se desea representar Ubicaciones y Caminos entre estas ubicaciones en forma de grafos direccionale, para esto se utilizan un conjunto de clases para facilitar esta tarea como por ejemplo:

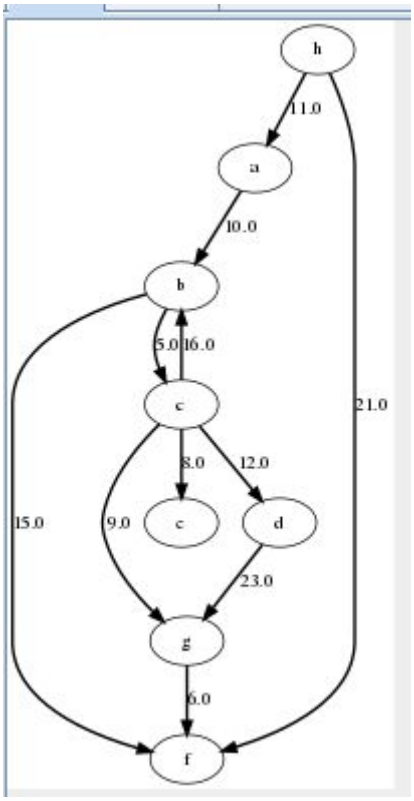
Ubicación:
Esta clase representa como su nombre lo indica las diferente ubicaciones que se encuentran en el gps, contiene atributos como nombre, un listado de *Caminos* y un tipo boolean que nos servirá para encontrar las *Rutas*;

```
public class Ubicacion {
    private String nombre;
    private ArrayList<Camino> caminos;
    private ArrayList<Camino> caminosAPie;
    private boolean visitado;
    public Ubicacion(String Nombre) {
        this.nombre = Nombre;
        caminos = new ArrayList<>();
        caminosAPie= new ArrayList<>();
    }
```

Camino:
Este representa las relaciones de una ubicación a otra, este camino es unidimensional ya que cuenta con una *Ubicación* Origen y Destino, además de atributos que caracterizan el definan al camino como distancia, tiempo desgaste y consumo de Gas.

```
public class Camino {
    private Ubicacion origen;
    private Ubicacion destino;
    private double distancia;
    private double tiempoEnVehiculo;
    private double tiempoAPie;
    private double consumoDeGas;
    private double desgastePersonal;
```

Estos dos objetos se construyen desde un archivo de entrada describiendo los caminos existentes, por ejemplo lo siguiente es una gráfica según el archivo de entrada a la par.




```
a|b|1.5|2|4.3|20|10
h|a|1.5|2|4.3|20|11
h|f|1.5|2|4.3|20|21
b|f|1.5|2|4.3|20|15
c|b|1.5|2|4.3|20|16
b|c|1.5|2|4.3|20|5
c|g|1.5|2|4.3|20|9
g|f|1.5|2|4.3|20|6
c|e|1.5|2|4.3|20|8
c|d|1.5|2|4.3|20|12
d|g|2|4|5|15|23
```

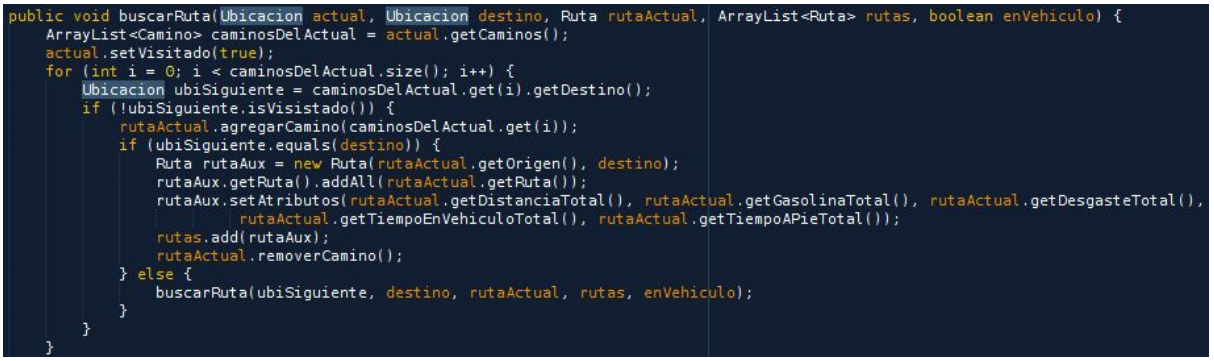
Rutas:
Uno de los principales objetivos del proyecto es proporcionarle información de las diferentes rutas al usuario, para lograr esto se una clase llamada Ruta que consiste en un conjunto de Caminos que conforman la ruta que llevan de una *Ubicación* Origen a uno Destino. Este objeto ya tendrá la cantidad de distancia, tiempo, desgaste y consumo de Gasolina Total, para la conveniencia del usuario.

```java
public class Ruta {

    private Ubicacion origen;
    private Ubicacion destino;
    private ArrayList<Camino> ruta;
    private double distanciaTotal;
    private double gasolinaTotal;
    private double desgasteTotal;
    private double tiempoEnVehiculoTotal;
    private double tiempoAPieTotal;
```

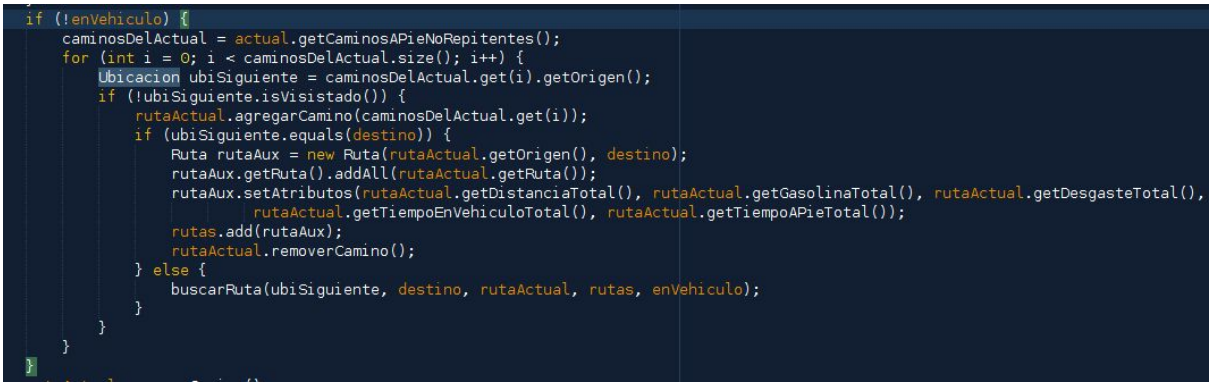
Cómo encontrar las *Rutas*:
Para ello se utiliza un algoritmo recursivo que trata de construir una *Ruta* (Lista De *Caminos*) empezando desde el Origen y usando todos sus caminos que llevan a otra

Ubicación y usar el metodo de nuevo y así hasta poder llegar al Destino.Al momento de llegar al Destino esa *Ruta* se guarda. Cuando se llega al Destino o a una *Ubicación* sin salida se regresa una Ubicación anterior con la ayuda de la recursividad para poder recorrer todos los caminos sin bucles.



```
public void buscarRuta(Ubicacion actual, Ubicacion destino, Ruta rutaActual, ArrayList<Ruta> rutas, boolean enVehiculo) {
    ArrayList<Camino> caminosDelActual = actual.getCaminos();
    actual.setVisitado(true);
    for (int i = 0; i < caminosDelActual.size(); i++) {
        Ubicacion ubiSiguiente = caminosDelActual.get(i).getDestino();
        if (!ubiSiguiente.isVisistado()) {
            rutaActual.agregarCamino(caminosDelActual.get(i));
            if (ubiSiguiente.equals(destino)) {
                Ruta rutaAux = new Ruta(rutaActual.getOrigen(), destino);
                rutaAux.getRuta().addAll(rutaActual.getRuta());
                rutaAux.setAtributos(rutaActual.getDistanciaTotal(), rutaActual.getGasolinaTotal(), rutaActual.getDesgasteTotal(),
                        rutaActual.getTiempoEnVehiculoTotal(), rutaActual.getTiempoAPieTotal());
                rutas.add(rutaAux);
                rutaActual.removerCamino();
            } else {
                buscarRuta(ubiSiguiente, destino, rutaActual, rutas, enVehiculo);
            }
        }
    }
```

A este algoritmo se le agrega si la búsqueda es a Pie ya que la dirección de los *Caminos* no importan.



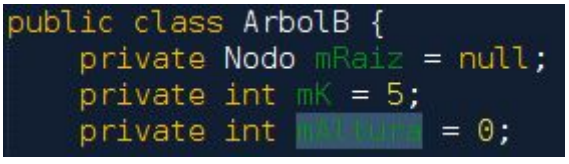
```
if (!enVehiculo) {
    caminosDelActual = actual.getCaminosAPieNoRepitentes();
    for (int i = 0; i < caminosDelActual.size(); i++) {
        Ubicacion ubiSiguiente = caminosDelActual.get(i).getOrigen();
        if (!ubiSiguiente.isVisistado()) {
            rutaActual.agregarCamino(caminosDelActual.get(i));
            if (ubiSiguiente.equals(destino)) {
                Ruta rutaAux = new Ruta(rutaActual.getOrigen(), destino);
                rutaAux.getRuta().addAll(rutaActual.getRuta());
                rutaAux.setAtributos(rutaActual.getDistanciaTotal(), rutaActual.getGasolinaTotal(), rutaActual.getDesgasteTotal(),
                        rutaActual.getTiempoEnVehiculoTotal(), rutaActual.getTiempoAPieTotal());
                rutas.add(rutaAux);
                rutaActual.removerCamino();
            } else {
                buscarRuta(ubiSiguiente, destino, rutaActual, rutas, enVehiculo);
            }
        }
    }
}
```

Mientras se van agregando Caminos a la Rutas se van sumando los atributos a los anteriores para tener la distancia Total y los demás.
Al tener todas las rutas se muestran en una panel que describe todo lo que necesita saber el Usuario.

ArbolB

Para el Árbol también se utilizaron varias clases para el facilitación de su creación e inserción de los Id de las Rutas. Cabe Mencionar que para este caso m = 5. Las clases que conforman la estructura son los siguientes:

ÁrbolB:
Aquí simplemente se guarda un Nodo que será nuestra raíz y se controla el tamaño del árbol en este caso la altura y m. Desde aquí también se insertan los IDs lo cual se explicará después.



```
public class ArbolB {
    private Nodo mRaiz = null;
    private int mK = 5;
    private int mAltura = 0;
```

Nodo:
Aqui guardaremos los objetos del árbol que en este caso son *Rutas* y agregamos encontramos punteros hacia otros *Nodos* los cuales hacen la estructura del Árbol.

```java
public class Nodo {
    int mK;
    int mB;
    Ruta[] mLlaves;
    Nodo[] mPunteros;
```

Split:
Al momento De Insertar Objetos al árbolB en ocasiones tendremos que aumentar la altura del Árbol para mantenerlo balanceado y para realizar eso usamos esta clase Split.

```java
public class Split {
    private Nodo mPuntero;
    private Ruta ruta;

        public Split(Nodo pPuntero, Ruta ruta) {
        this.mPuntero = pPuntero;
        this.ruta = ruta;
    }
```

Cómo se insertan Objetos al Árbol:
Primero se verifica si existe un nodo Raíz Verificando si la altura es 0, si no existe se crea uno con el elemento que se desee insertar, en ese momento el mb del nodo será 1. Ahora si ya existe un nodo de ese tipo se va directamente al insert que verifica si es necesario un split

```java
public void insert(Ruta ruta) {
    if (this.mAltura == 0) {
        this.mRaiz = new Nodo(this.mK, ruta);
        this.mAltura = 1;
        return;
    }

    Split splitted = insert(this.mRaiz, ruta, 1);

    if (splitted == null) {
    } else {

        Nodo ptr = this.mRaiz;

        this.mRaiz
                = new Nodo(this.mK, splitted.getRuta());
        this.mRaiz.mPunteros[0] = ptr;
        this.mRaiz.mPunteros[1] = splitted.getmPuntero();
        this.mAltura++;
    }
}
```

En la función insert se utiliza recursividad para insertar el elemento ya que este método tiene como parámetro el Nodo que se está evaluando y el nivel eso va ir cambiando conforme la recursividad mientras que el parámetros ruta será el mismo siempre ya que siempre será el mismo objeto que se desee insertar. Primero se encuentra la variable i que representa la posición del Nodo Actual que se está evaluando. Si el nivel a evaluar es el mismo que la altura y existe un espacio para la ruta simplemente se inserta el nodo. Y si no se debe de llamar la funcion de nuevo pero con el Nodo hijo del Actual según las comparaciones del id.

```
protected Split insert(Nodo node, Ruta ruta, int level) {
    Split splitted = null;
    Nodo ptr = null;
    int i = 0;
    while ((i < node.mB) && (ruta.getId() > (node.mLlaves[i]).getId())) {
        i++;
    }
    if ((i < node.mB) && ruta.getId() == (node.mLlaves[i]).getId()) {
        node.mLlaves[i] = ruta;
        return null;
    }
    if (level < this.mAltura) {
        splitted = insert(node.mPunteros[i], ruta, level + 1);
        if (splitted == null) {
            return null;
        } else {
            ruta = splitted.getRuta();
            ptr = splitted.getmPuntero();
        }
    }
    i = node.mB - 1;
    while ((i >= 0)
            && (node.mLlaves[i] == null || ruta.getId() < (node.mLlaves[i]).getId())) {
        node.mLlaves[i + 1] = node.mLlaves[i];
        node.mPunteros[i + 2] = node.mPunteros[i + 1];
        i--;
    }
    node.mLlaves[i + 1] = ruta;
    if (splitted != null) {
        node.mPunteros[i + 2] = splitted.getmPuntero();
    }
```

Luego de hacer el insert se verifica si el número de Rutas en el nodo Actual contiene más de lo que debería (en este caso más de 4) y se es asi se hace el split del árbol, en este caso la función devuelve un split, aumentando la altura del árbol.

```
node.mB++;
if (node.mB > 2 * mK) {
    Nodo newnode = new Nodo(this.mK);
    newnode.mPunteros[this.mK] = node.mPunteros[node.mB];
    node.mPunteros[node.mB] = null;
    node.mB = this.mK + 1;
    for (i = 0; i < this.mK; i++) {
        newnode.mLlaves[i] = node.mLlaves[i + node.mB];
        node.mLlaves[i + node.mB] = null;
        newnode.mPunteros[i] = node.mPunteros[i + node.mB];
        node.mPunteros[i + node.mB] = null;
    }
    node.mB--;
    splitted= new Split(newnode, node.mLlaves[node.mB]);
    node.mLlaves[node.mB] = null;
    newnode.mB = this.mK;
    node.mB = this.mK;
    return splitted;
}

return null;
```

Como se elimina en el ArbolB:

Primero se encuentra el nodo en el que se encuentra el ID que se desea borrar, para ello se utiliza un algoritmo de búsqueda en base a recursividad básica. Luego de encontrarlo se elimina. Ahora el algoritmo se divide en dos partes con un condición el cual verifica si el nodo tienen hijos o no.
Si tiene hijos se busca el valor menor a la derecha con otro método de recursividad para que luego este valor sea asignado al espacio en donde se borro el elemento y se elimina de su nodo origen. El nodo de donde se encontró el menor a la derecha se le hace la verificacion si culple con tener elementos mayor que mK(en este caso 2).
Si es menor se verifica la posibilidad de prestarle un elemento a su hermano y si no es así se deben de fusionar los nodos para que siga cumpliendo con lo deseado.

```
public void eliminar(int id) {
    try {
        int nivel=1;
        Nodo padre=mRaiz;
        int indice=0;
        Nodo nodo = buscarElemento(mRaiz, id,nivel,padre,indice);
        for (int i = 0; i < nodo.mLlaves.length; i++) {
            if (nodo.mLlaves[i] != null && nodo.mLlaves[i].getId() == id) {
                nodo.mLlaves[i] = null;
                nodo.mB--;
            }
        }
        if (nivel<mAltura) {
            Nodo menorALADerecha=obtenreMayorALaDerecha(padre.mPunteros[indice+1],padre);
            nodo.mLlaves[indice]=menorALADerecha.mLlaves[0];
             if (menorALADerecha.mB<this.mK) {
                if (indice==0) {
                    if (padre.mPunteros[1].mB>this.mK) {
                        padre.mLlaves[indice]=prestar(menorALADerecha, padre.mPunteros[1], padre.mLlaves[0]);
                    }else{
                        padre.mPunteros[0]=fusionarNodos(menorALADerecha, padre.mPunteros[1], padre.mLlaves[0]);
                        padre.mLlaves[indice]=null;
                    }
                }else{
                    if (padre.mPunteros[indice-1].mB>mK) {
                        padre.mLlaves[indice]=prestar(menorALADerecha, padre.mPunteros[indice-1], padre.mLlaves[indice]);
                    }else if(padre.mPunteros[indice+1].mB>mK){
                        padre.mLlaves[indice]=prestar(menorALADerecha, padre.mPunteros[indice+1], padre.mLlaves[indice]);
                    }else{
                        padre.mPunteros[indice]=fusionarNodos(menorALADerecha, padre.mPunteros[indice-1], padre.mLlaves[indice]);
                        padre.mLlaves[indice]=null;
                    }
                }
            }
```

La otra división del algoritmo es si el nodo no tiene hijos en donde inmediatamente se verifica si aún cumple con ser mayor o igual que mK. Si no es así también debe de prestarle a su hermano si es posible o fusionarse con el mismo.

```
}else{
    if (nodo.mB<this.mK) {
        if (indice==0) {
            if (padre.mPunteros[1].mB>this.mK) {
                padre.mLlaves[indice]=prestar(nodo, padre.mPunteros[1], padre.mLlaves[0]);
            }else{

                padre.mPunteros[0]=fusionarNodos(nodo, padre.mPunteros[1], padre.mLlaves[0]);
                padre.mLlaves[indice]=null;
            }
        }else{
            if (padre.mPunteros[indice-1].mB>mK) {
                padre.mLlaves[indice]=prestar(nodo, padre.mPunteros[indice-1], padre.mLlaves[indice]);
            }else if(padre.mPunteros[indice+1].mB>mK){
                padre.mLlaves[indice]=prestar(nodo, padre.mPunteros[indice+1], padre.mLlaves[indice]);
            }else{
                padre.mPunteros[indice]=fusionarNodos(nodo, padre.mPunteros[indice-1], padre.mLlaves[indice]);
                padre.mLlaves[indice]=null;
            }
        }

    }
```