# Universidad de los Andes

Ingeniería de Sistemas y Computación
ISIS1206 – Estructura de Datos
Proyecto 1 – Entrega final

| **Integrantes** | |
|---|---|
| Laura Maya (Parte B) | 201423854 |
| Carlos Peñaloza (Parte A) | 201531973 |
| **Repositorio Bitbucket** | |
| https://bitbucket.org/proyecto_201620_sec_1_team_14/proyecto_1_201620_sec_1_team_14 | |

## Introducción

Realice una breve explicación de que hicieron y cómo llegaron a la solución propuesta.

La primera etapa para las dos primeras partes del proyecto, consistió en implementar la rutina de lectura de archivos JSON en la cual se utilizó la librería Gson para poder importar los datos del archivo como objetos dentro del mundo de nuestra aplicación. Esta información fue almacenada en listas sencillamente encadenadas, se crearon dos listas para cada parte, una para la información de todas las colisiones, y otra en la cual se encontraba la información de las entradas con más de cuarenta colisiones slight y cien colisiones serious respectivamente. Esto se realizó para facilitar la generación del primer reporte de cada punto en un tiempo menor.

**Parte A**
Con el fin de organizar la información de manera adecuada, creamos Local Authorities, Wards y Años. El sistema de SmartCities tendrá una lista de Local authorities, cada Local Authority guardará una lista de Wards cada Ward guardará una lista de entradas de colisiones por años en donde se guardará la información de todos los tipos de colisiones junto con los nombres del Ward y Local authority para facilitar la búsqueda. De esta manera las local authorities tienen la información de sus colisiones organizada de tal forma que retornar alguna información al sistema sea sencillo.

**Parte B**
Para poder organizar la información de manera adecuada, creamos Áreas y Años. El sistema de SmartCities tendrá una lista de Áreas, cada área guardará su nombres y una lista de años en donde se guardará la información de las colisiones de cada uno. De esta manera las áreas pueden manipular sus propios años antes de retornar alguna información al sistema, y este solo deberá pedir a las Áreas la información para obtener la información de una sola área.

**Parte C**
Para la parte C, primero leímos la información del archivo Ingreso_al_parqueadero.csv , para obtener la información de los carros que entraban e irlos metiendo en la cola de espera en el orden en que entraban. Una vez importamos todos los carros leímos el archivo Salida_del_parqueadero.csv para

poder saber en qué posición debía salir cada carro y en base a esto poderlos parquear de manera ordenada minimizando la cantidad de movimientos a las hora de sacarlos.

## Requerimientos Funcionales

Identifique los 3 requerimientos funcionales principales para el sistema propuesto y documentelos siguiendo el siguiente formato:

**Parte A & B**

| | |
|---|---|
| **Nombre** | Importar la información de los archivos Json |
| **Resumen** | Importar y guardar en el sistema la información de los archivos Json |
| **Entradas** | |
| Archivos con la información | |
| **Resultados** | |
| La información del archivo queda guardada en el sistema en listas. | |

| | |
|---|---|
| **Nombre** | Procesar las solicitudes de los alcaldes y generar los reportes pertinentes a estas |
| **Resumen** | Procesa la información guardada y en base a estas atiende las solicitudes de los alcaldes |
| **Entradas** | |
| Solicitud del alcalde | |
| **Resultados** | |
| Se imprime en consola la información solicitada por los alcaldes. | |

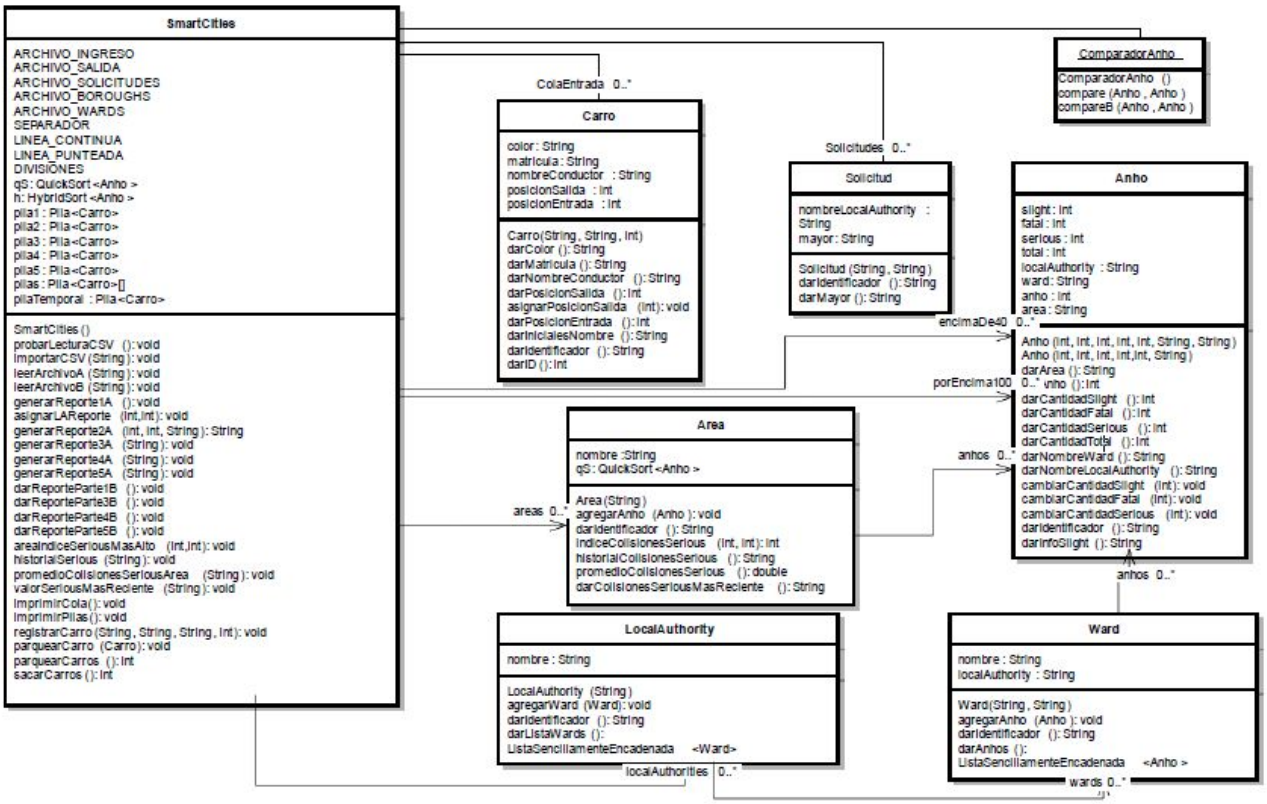
**Parte C**

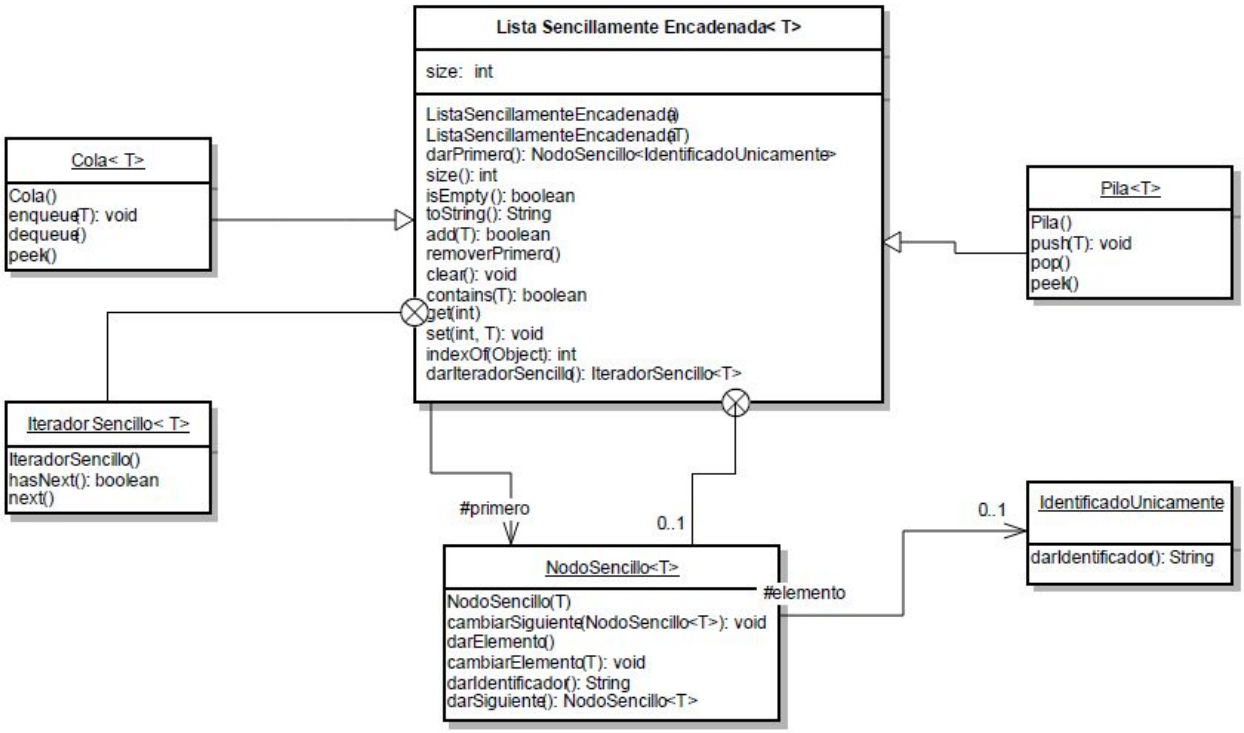
| | |
|---|---|
| **Nombre** | Minimizar los movimientos de salida de los carros del parqueadero |
| **Resumen** | Se conoce el orden de entrada y el orden de salida de los carros. Con base en esto se deben tratar los carros de manera que el número de movimientos para sacar los carros sea minimizado. |
| **Entradas** | |
| Archivo Ingreso_al_parqueadero.csv y Salida_Del_Parqueadero.csv | |

| **Resultados** |
| --- |
| Los carros quedan almacenados en una cola y cada carro sabe en qué posición entró y en qué posición debe salir. Una vez todos los carros están en la cola se van parqueando en el orden de salida para que los movimientos a la hora de sacar los carros sea igual al número de carros que hay en el parqueadero. |

## Modelo del mundo

Presente de forma clara su modelo del mundo usando la notación UML y explique los diferentes elementos del qué consiste (para cada parte del proyecto).






## Diseño, implementación y pruebas de las estructuras de datos

Indique cómo **las estructuras que usted creó,** las cuales deben expresarse en el modelo UML, satisfacen los requerimientos funcionales propuestos por el cliente al igual como los no funcionales.

El uso de listas sencillamente encadenadas nos permitió organizar los datos que llegan del archivo Json, de una manera más eficiente ya que estas no tienen un tamaño definido, por el contrario el tamaño se va adaptando a las necesidades que tengamos. Además no necesita tener el espacio consecutivo en la memoria ya que puede buscar donde hay espacio y almacenar ahí los datos, por lo cual resulta menos costosa en espacio. También, es importante resaltar que en este tipo de estructura de datos es fácil realizar la implementación de inserción y borrado de elementos. Por último, esta estructura permite satisfacer los requerimientos funcionales propuestos ya que su versatilidad permite la implementación de colas y pilas, estructuras cuyas propiedades eran necesarias.

Para manejar los datos, las colas y las pilas son estructuras que nos permitieron organizar mejor los procesos que tenemos que hacer, la ventaja principal de la cola es que se pueden agregar y remover elementos fácilmente (al comienzo y al final), lo cual nos permite guardar las peticiones de manera en que podían ser resueltas en el orden de llegada, y con las pilas pudimos guardar temporalmente los datos cuando necesitemos ordenar o buscar un elemento en particular de una forma eficiente en tiempo, además de ser la estructura más importante en la implementación del punto C.

**Indique qué cambios se realizaron desde la primera entrega y el porqué de ellos**.

Desde la primera entrega fue necesario crear una clase comparadora de años para así poder ordenar las entradas de colisiones teniendo en cuenta más de un criterio para poder realizar las consultas en un tiempo menor.

También fue necesario agregar atributos a los “anhos” (registros de colisiones) para que guardaran el nombre de su area, autoridad local o ward con el fin de realizar la búsqueda con facilidad.

## Análisis de complejidad

De los requerimientos propuestos tome los 3 más importantes, copie el fragmento de código que satisface dicho requerimiento y haga un análisis de complejidad para cada uno.

**Lectura archivo JSON**

```
public void leerArchivoA(String ruta)
{
	File f = new File(ruta);
	JSONTokener tkn = new JSONTokener(new FileReader(f));
	JSONArray arr = new JSONArray(tkn);

	for(int i = 0; i < arr.length(); i++)
	{
		JSONObject actual = arr.getJSONObject(i);
		JSONArray info = actual.names();
		String localAuthority = actual.getString("Local Authority");
		String nombreWard = actual.getString("Ward name");
		Ward nuevoWard = new Ward(nombreWard, localAuthority);
		for(int j = 0; j< info.length(); j++)
		{
			String txt = info.getString(j);
			if(txt.startsWith("Slight")){
				int anho = Integer.parseInt(txt.substring(txt.length()-4,txt.length()));
				int slight = (int) actual.get("Slight "+ anho);
				int fatal = actual.getInt("Fatal "+ anho);
				int serious = actual.getInt("Serious "+ anho);
				int total = actual.getInt("Total "+ anho);

				Anho nuevoAnho = new Anho(anho, slight, fatal, serious, total, nombreWard,
				localAuthority);
				nuevoWard.agregarAnho(nuevoAnho);

				if(slight >= 40)
				{
					encimaDe40.add(nuevoAnho);
				}
			}
```

```
                    }
                    ListaSencillamenteEncadenada<LocalAuthority>.IteradorSencillo<LocalAuthority> it =
                    localAuthoritiesA.darIteradorSencillo();
                    boolean encontro = false;
                    LocalAuthority nuevo = null;

                    while(it.hasNext() && !encontro)
                    {
                            nuevo = it.next();

                            if(nuevo.darIdentificador().equals(localAuthority)){
                                    nuevo.agregarWard(nuevoWard);
                                    encontro = true;
                            }
                    }
                    if(!encontro){
                            nuevo = new LocalAuthority(localAuthority);
                            nuevo.agregarWard(nuevoWard);
                            localAuthoritiesA.add(nuevo);

                    }
            }
    }
```

Este método recorre cada objeto de un JSON array, y dentro de este recorrido ocurren dos recorridos: en el primero cada objeto es partido en un array de sus elementos, y en el segundo se revisa si la local authority en el objeto actual ya existe en la lista de local authorities. Los dos recorridos internos se pueden aproximar a menos de O(2N) es decir O(n) pero al encontrarse anidados con el recorrido externo, la complejidad de este algoritmo es O(n^2). Esta es la solución óptima ya que para leer todos los datos es necesario realizar los recorridos previamente mencionados, y además al agregar de una vez los registros con accidentes slight mayores a cuarenta, se consigue agilizar la generación del primer reporte.

**Ejemplo consulta**

```
/**
 * Imprime en la consola el promedio de colisiones serious del area que entra por parametro
 * @param nomArea Nombre del area del cual se quiere conocer el promedio
 */

public void promedioColisionesSeriousArea (String nomArea)
{
        Area actual = null;
        ListaSencillamenteEncadenada<Area>.IteradorSencillo<Area> ite = areas.darIteradorSencillo();
        boolean encontro = false;
        while(ite.hasNext() && !encontro)
        {
                actual = ite.next();
                if(actual.darIdentificador().equalsIgnoreCase(nomArea))
                {
```

```
                        System.out.println("El promedio de colisiones del area " + actual.darIdentificador() + " es: "
                        + actual.promedioColisionesSerious());
                            encontro = true;
                    }
                }
                if(!encontro)
                        System.out.println("No existe el area ingresada");
        }

/**
         * Retorna el promedio de colisiones que tiene el area por año
         * @return Promedio de colisiones Serious del area por año
         */
        public double promedioColisionesSerious()
        {
                qS.sortB(anhos);
                int suma = 0;
                ListaSencillamenteEncadenada<Anho>.IteradorSencillo<Anho> ite = anhos.darIteradorSencillo();
                Anho actual = null;
                while(ite.hasNext())
                {
                        actual = ite.next();
                        suma += actual.darCantidadSerious();
                }
                return suma/anhos.size();
        }
```

El método anterior en la primera parte busca el área que le entra por parámetro dentro de la lista de áreas que tiene y una vez lo encuentra le pide a esa área que le devuelva el promedio de colisiones que tiene. Esto quiere decir que la complejidad del método es de O(n^2) ya que debe recorrer las áreas y en esa área debe recorrer todos los años para poder encontrar el promedio de colisiones de esa área. Esta no es la solución más óptima ya que en lugar de recorrer los años dentro del área para conocer el promedio, podríamos tener un atributo en el área en el cual el número total de colisiones se fueran sumando a medida que se va agregando un nuevo año, y siempre se podría tener ese dato, convirtiendo la complejidad del método en O(n)

**Parquear los carros**

```
        /**
         * Parquea todos los carros que estan en la cola de espera
         * @return Cantidad de movimientos que realizo para parquear todos los carros en orden
         */
        public int parquearCarros(){
                int posicion = colaEntrada.size();
                int movimientos = 0;
                while(colaEntrada.size() > 0 && posicion>0){
                        Carro act = colaEntrada.dequeue();
                        movimientos++;
                        if(act.darPosicionSalida() == posicion){
                                parquearCarro(act);
                                posicion--;
```

```
                    }
                    else{
                        colaEntrada.enqueue(act);
                        movimientos ++;
                    }
                }
                return movimientos;
        }

/**
         * Busca un parqueadero con cupo dentro de los 3 existentes y parquea el carro
         * @param aParquear es el carro que se saca de la cola de carros que estan esperando para ser parqueados
         */
        public void parquearCarro(Carro aParquear)
        {
                boolean espacio = false;
                for(int i = 0; i<pilas.length && !espacio; i++ )
                {
                    if(pilas[i].size() < 4)
                    {
                        pilas[i].push(aParquear);
                        espacio = true;
                    }
                }
        }
```

Este método saca un carro de la cola y si su posición de salida coincide con la posición que estan parqueando en este momento manda a parquear el carro, y el método de parquear el carro busca la primera posición disponible en las pilas. Por ende en el peor caso (parquear el último carro) la complejidad del algoritmo es de O(n^2) ya que si es el último carro ya recorrió toda la cola, y tendría que recorrer todas las pilas para encontrarle lugar en el parqueadero, el mejor caso sería que el primer carro que buscamos está en la posición de salida (complejidad O(1)) ya que no recorreremos las pilas y obtendremos el carro en el primer carro que pidamos. No es la solución más óptima ya que podriamos juntar los métodos y así ir sabiendo que pila ya esta llena y por ende ir guardando en la pila que sigue y así no tener que recorrer las pilas cada vez que se va a parquear un carro.