Universidad de los Andes

Ingeniería de Sistemas y Computación
ISIS 1206 – Estructura de Datos
Proyecto 2 – Entrega final

Integrantes	
Laura Maya (Parte A, C2-2, C3)	201423854
Carlos Peñaloza (Parte B, C1,C2-1)	201531973
Repositorio Bitbucket	
https://bitbucket.org/proyecto_201620_sec_1_team_14/proyecto_2_201620_sec_1_team_14	

Introducción

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

Para llegar a la solución de las tres partes, primero tuvimos que implementar una lectura de archivos JSON usando la librería GSON para guardar esta información en las diferentes estructuras de datos usadas en el proyecto. La información se guardó en tablas de hash (Linear probing y Separate Chaining), colas de prioridad orientadas a máximo, y en árboles rojos negros. Con el fin de organizar la información creamos las clases Local Authority, y Tablas de Flujo; la primera es utilizada principalmente para las partes 1a, 1b, 2a y 2b y la segunda para la tercera parte de los puntos 1 y 2. Para poder mostrar las gráficas que comparan los comportamientos de las consultas en la parte C del proyecto, creamos una clase llamada Gráfica y usamos la librería JFREECHART.

Parte A
Con el fin de organizar la información de manera eficiente creamos un heap orientado a la mayor prioridad que guarda todas las local authorities, eso facilita generar el primer reporte ya que por medio del algoritmo heapsort podemos imprimir de manera eficiente todas las local authorities ordenadas por su prioridad. Usamos también una tabla de hash usando el método separate chaining que en su valor guarda el código de cada local authority y en su valor contiene una tabla de hash usando el método linear probing para guardar la información de los diferentes años de los cuales se tiene información. De esta manera consultar la información de cierta local authority en cierto año se realiza de manera más rápida y directa. Para la tercera parte guardamos en un árbol rojo-negro los nombres de las local authorities (llave) y asociado a estos el código de cada una (valor), para poder consultar el código y posteriormente en un árbol rojo-negro que guarda el código (llave) y la información en tablas de flujo (valor) buscar la información de flujo de carros de cada local authority. Estas estructuras nos permiten consultar el flujo de carros y de vehículos totales en un año para un local authority por su nombre.

Parte B
En la parte B se debían llevar a cabo las mismas consultas pero utilizando estructuras diferentes, en la práctica no es muy diferente el resultado pero al analizar los tiempos de

ejecución, se pueden identificar diferencias que podrían ser significativas si la cantidad de datos creciera en grandes cantidades.

Parte C

Para la parte C creamos archivos que en lugar de local Authorities tuvieran ciudades y usamos los atributos que ya habíamos creado para la parte 1 (Heap). Con estos importamos estos nuevos archivos realizamos la primera solicitud de la parte C. Para la segunda parte añadimos un atributo a las tablas de hash que guarde el número de colisiones que tuvo cada una a la hora de guardar los datos y estas las usamos para comparar el comportamiento de ambas tablas de hash. Para todas las solicitudes de esta parte guardamos la información de los tiempos y/o las colisiones en tablas de datos para usar la clase Gráfica y que esta muestre los gráficos pertinentes cuando se realicen las solicitudes.

Requerimientos Funcionales

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

Partes A & B

Nombre	R1. 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 tablas de hash, heaps y árboles rojos y negros.	

Nombre	R2. Consultar las autoridades locales ordenadas por su prioridad
Resumen	Teniendo ordenado el heap de los local authorities de mayor a menor prioridad, realiza heap sort para imprimir la información de cada uno.
Entradas	
Resultados	
Se imprime en consola la información de las autoridades locales	

Laura Maya 201423854

Carlos Peñaloza 201531973

Nombre	R3. Consultar el número de accidentes en un local authority en un año.
Resumen	Dado un año y un local authority, se imprime en consola la cantidad de accidentes que se reportaron en ese año y ese local authority.
Entradas	
El año y el local authority del cual se quiere conocer la información	
Resultados	
Se imprimió en consola la cantidad de accidentes ocurridos en el año y la local authority que entró por parámetro	

Nombre	R4. Consultar el flujo de carros y de vehículos en general en un año y un local authority
Resumen	Busca el flujo de carros y flujo de todos los vehículos en un local authority en algún año
Entradas	
Nombre del local authority y año del que se quiere conocer la información	
Resultados	
Se imprime en consola el flujo de carros y de todos los vehiculos del año y el local authority solicitado	

Parte C

Nombre	R1. Comparar heap con lista y con arreglo
Resumen	Se compara el tiempo de ejecución del heap implementado con lista y con arreglo cambiando la cantidad de elementos. Para esto se llevan a cabo las consultas con cien datos diferentes y se calcula el tiempo promedio.
Entradas	
Resultados	
Se muestra el tiempo promedio de ejecución de ambos heaps en una gráfica utilizando JFreeChart.	

Laura Maya 201423854

Carlos Peñaloza 201531973

Nombre	R2. Se comparan las dos formas de hacer la segunda consulta.
Resumen	Se ejecuta 100 veces los puntos 2A y 2B y se obtiene el tiempo promedio de cada uno para compararlos.
Entradas	
Resultados	
Se conoce el tiempo promedio de cada método.	

Nombre	R3. Comparar el número de colisiones cambiando el tamaño inicial de las tablas de hash
Resumen	Se inicializan las tablas de hash con diferentes valores y se registran el número de colisiones que ocurren en cada caso. Se considera colisión solamente que la casilla que debería ocupar esté ya ocupada por otro dato mas no los intentos que tiene que hacer para llegar a la posición que va a ocupar (Linear probing) o para insertarse en la lista que se extiende de la casilla (Separate Chaining). Luego se buscan todas las autoridades locales y se toma un tiempo promedio que demora cada tabla en hacer una búsqueda.
Entradas	
Resultados	
Se conoce el número de colisiones y el tiempo promedio de búsqueda de cada intento y se muestran sus respectivas gráficas	

Nombre	R4. Comparar tiempos tercera consulta
Resumen	Se comparan los tiempos de ejecución de la tercera consulta en la parte A y la parte B. Se ejecutan las consultas 100 veces para obtener el tiempo promedio de estas y poder compararlas apropiadamente.
Entradas	
Resultados	

Laura Maya 201423854

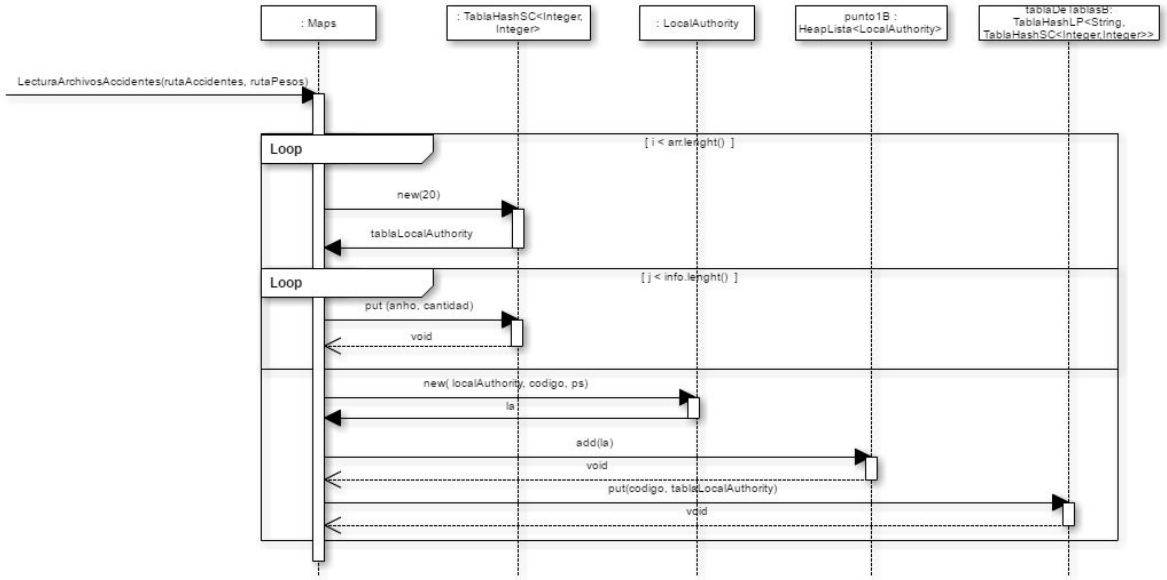
Carlos Peñaloza 201531973

Se conocen y comparan los tiempos de ejecución de la tercera consulta por ambos métodos. Se grafican los resultados para mayor claridad.

Modelo del mundo

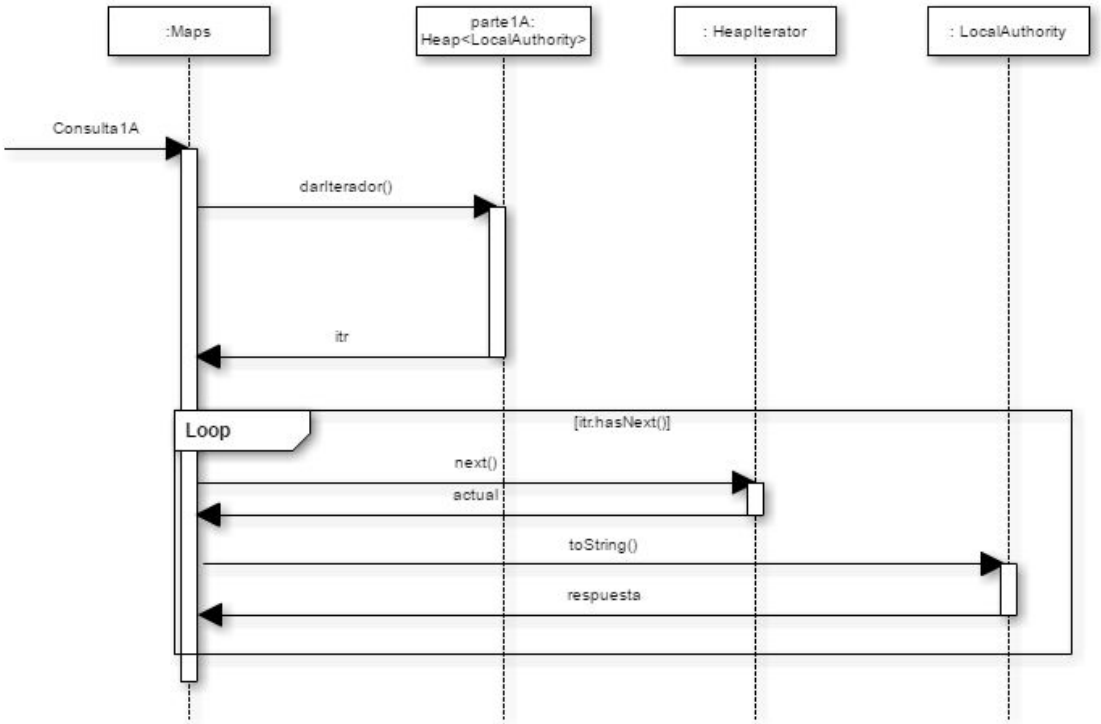
Presente de forma clara su modelo del mundo usando la notación UML y explique los diferentes elementos del qué consiste (para cada parte del proyecto).

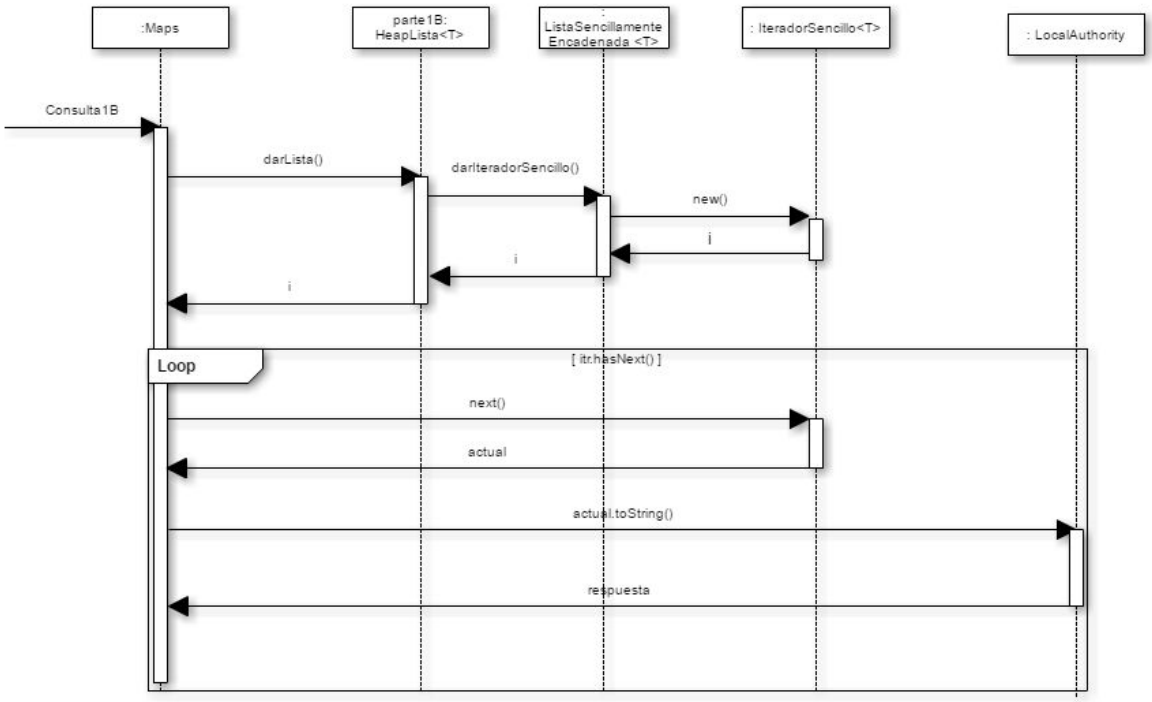
Diagramas de Secuencia



Laura Maya 201423854

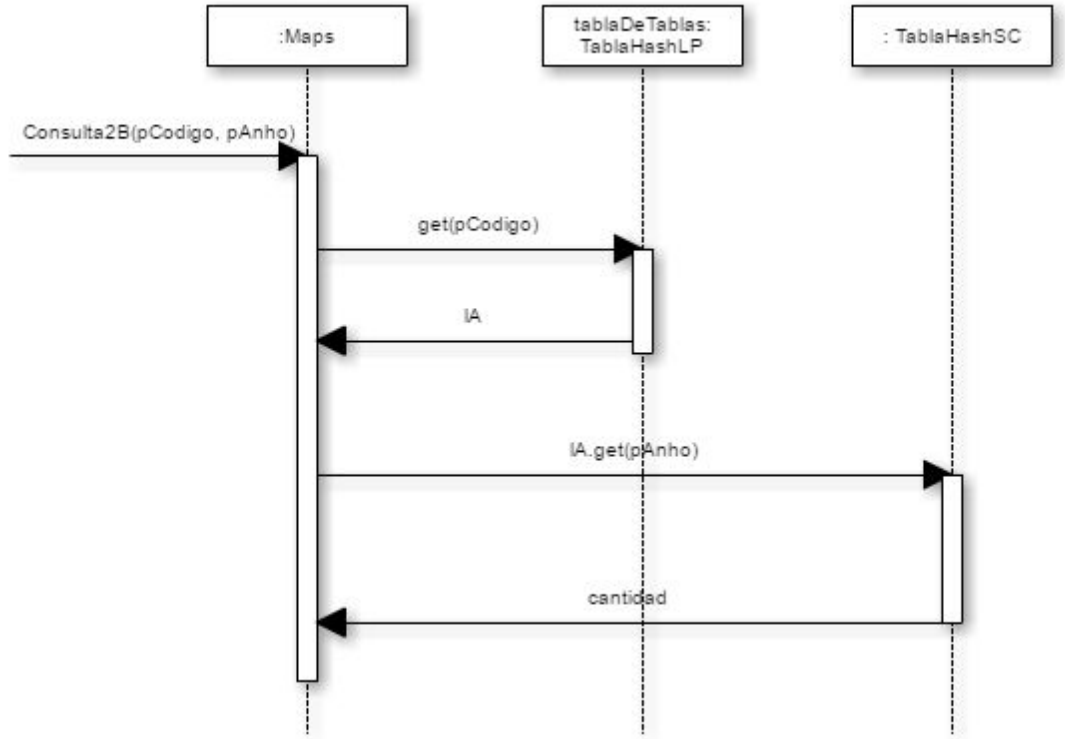
Carlos Peñaloza 201531973

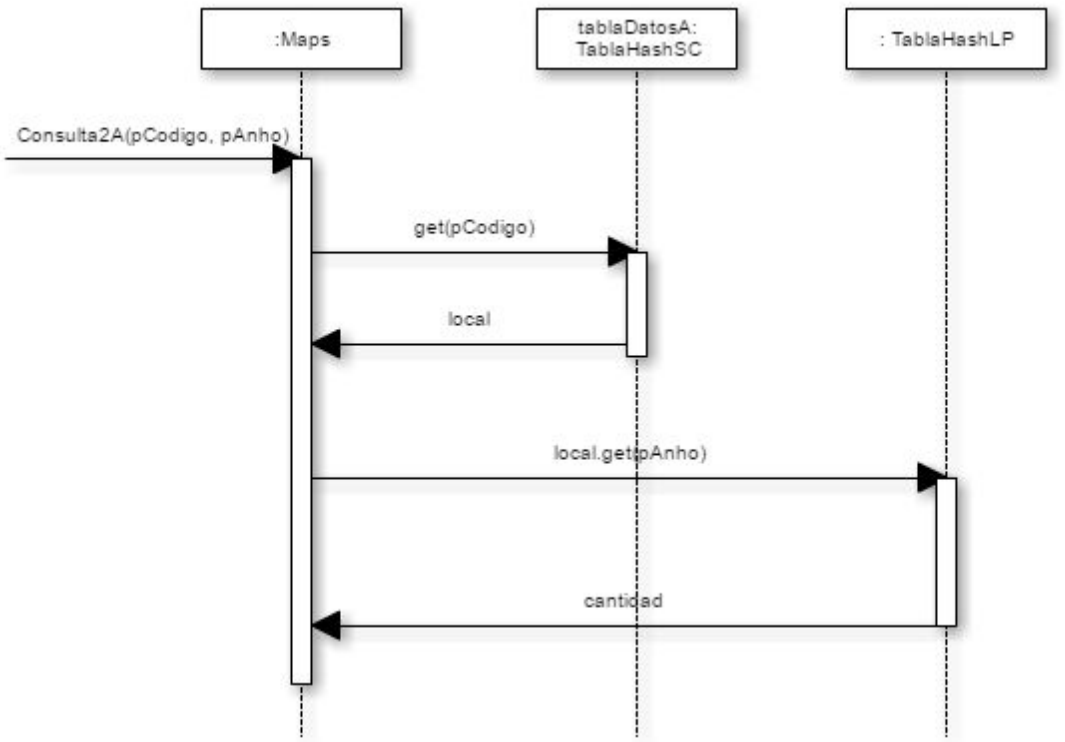




Laura Maya 201423854

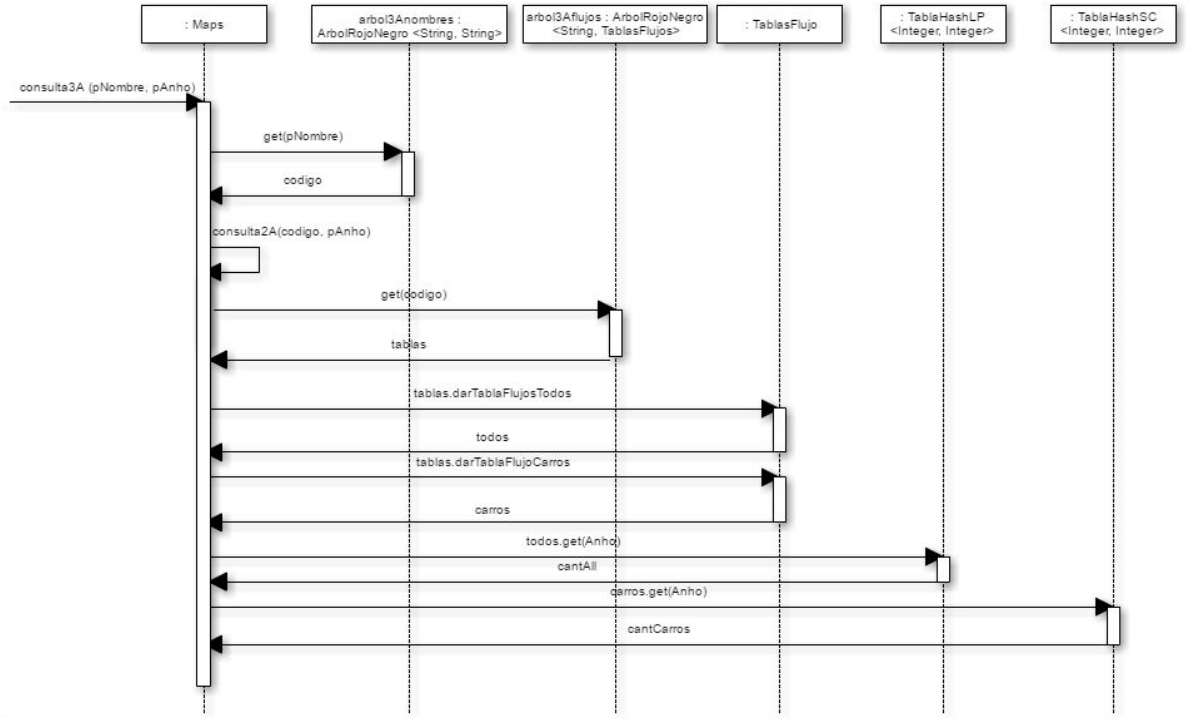
Carlos Peñaloza 201531973

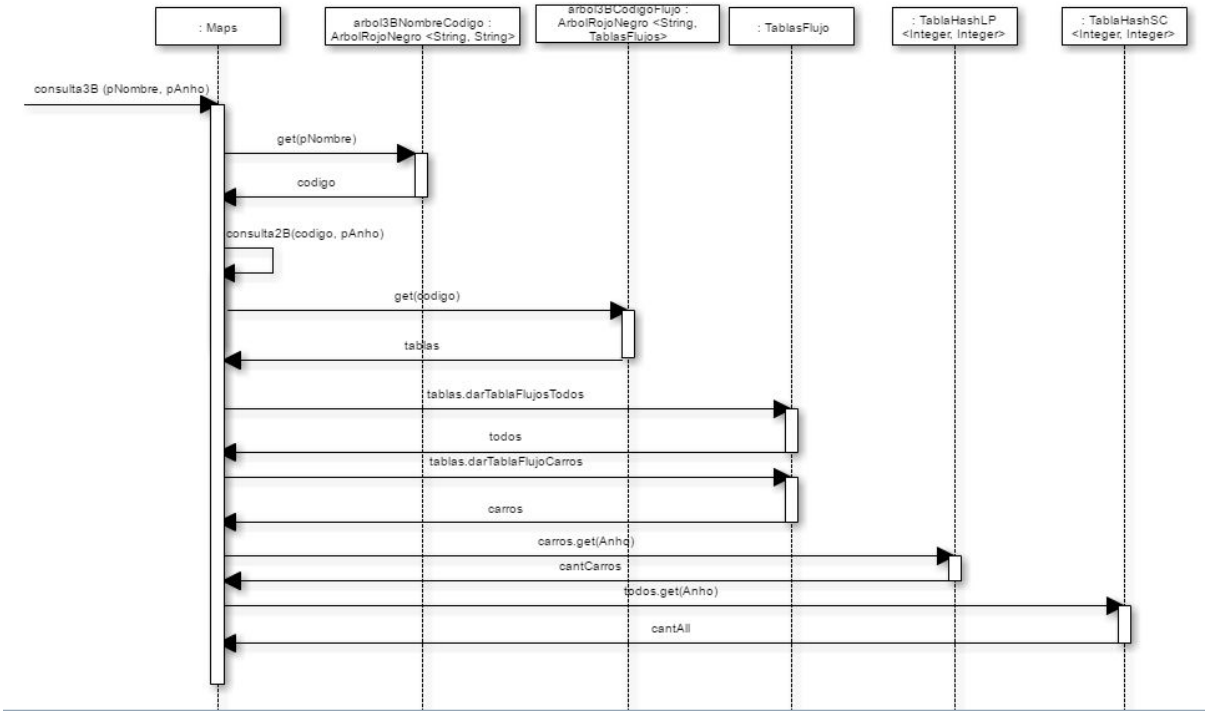




Laura Maya 201423854

Carlos Peñaloza 201531973

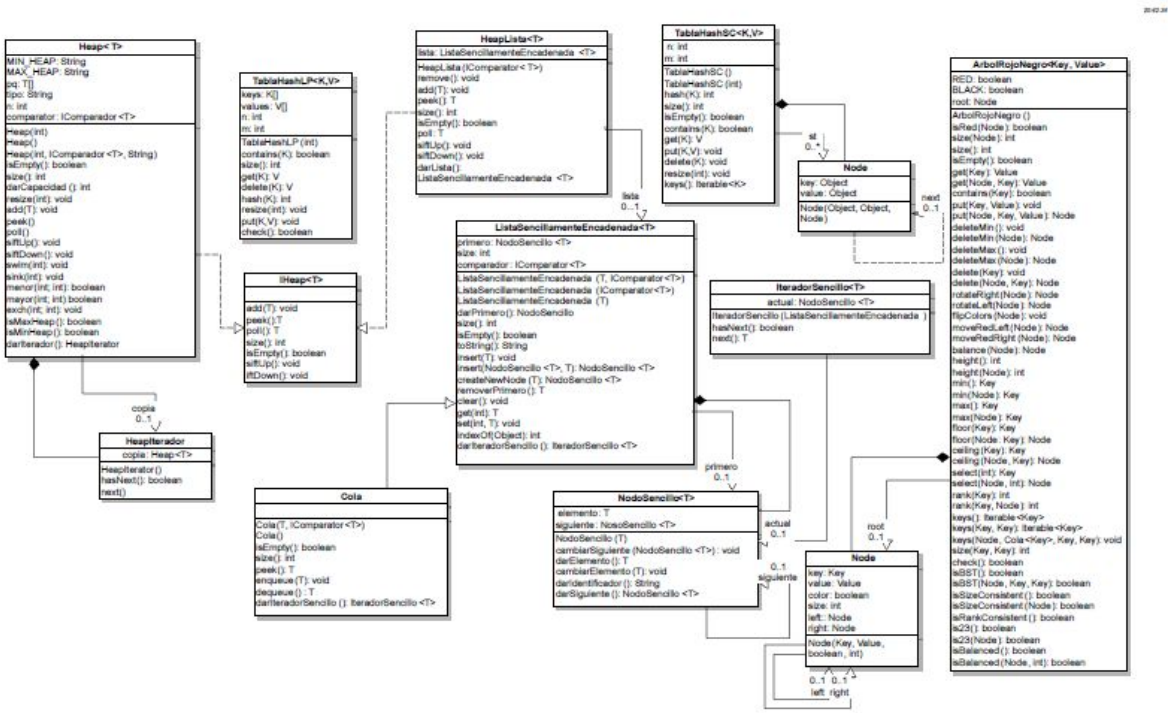


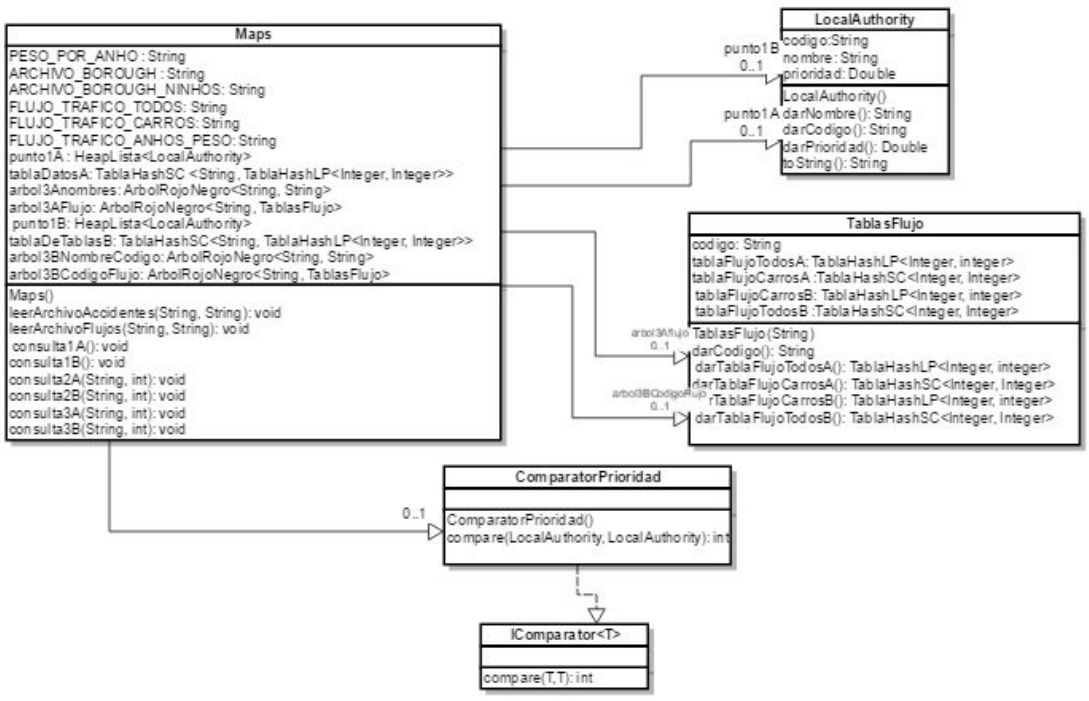


Diagramas UML

Laura Maya 201423854

Carlos Peñaloza 201531973





Laura Maya 201423854

Carlos Peñaloza 201531973

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.

Tabla de Hash: La utilización de tablas de Hash es útil para el proyecto ya que estas permiten buscar e insertar datos con una complejidad temporal de O(1), sin importar el tamaño de la estructura, siempre y cuando se utilice una función de Hash adecuada.

Cola de prioridad: Las colas de prioridad son importantes ya que estas permiten ordenar la lista según se desee (mayor a menor o viceversa) mientras se están leyendo y agregando los datos, lo cual reduce la complejidad temporal requerida en el proyecto anterior al tener que leer y ordenar por aparte antes de poder procesar los datos.

Árbol rojo negro: Las operaciones de insertar, remover y obtener elementos tienen complejidad de O(log(n)). Es una estructura que se balancea automáticamente por lo cual se garantiza que la complejidad temporal no cambiará. Es particularmente útil cuando se deben insertar o eliminar datos frecuentemente, y además ofrece todas las ventajas de un árbol 23 con la facilidad de uso de un árbol de búsqueda binario.

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

Desde la primera entrega le agregamos un atributo a las tablas de Hash que guarda el número de colisiones totales que tiene cuando ha intentado agregar los datos, y respectivamente el método que retorna esta cantidad. Para Linear Probing se considera colisión cada vez que intenta guardar el dato en una posición del arreglo y este está ocupado, mientras que en Separate Chaining solo se considera colisión cuando choca por primera vez con el arreglo si ya hay elementos en la posición deseada pero no considera colisiones la cantidad de elementos que hay dentro de la lista en cada posición.
Para poder cumplir con el la solicitud 2c-2 le agregamos un constructor a la clase principal (Maps) en el cual entran como parámetros el tamaño inicial de las tablas de hash.
Además se agregó una clase llamada Gráfica que es la que permite graficar los resultados de la parte C cuando se consultan las diferentes solicitudes.

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.

Laura Maya 201423854

Carlos Peñaloza 201531973

La lectura de datos, tiene una complejidad temporal de O(n^2), puesto que los archivos de flujo y accidentes deben ser leídos en su totalidad para guardar cada dato, lo cual implica hacer dos recorridos, con el fin de guardar los datos en la primera columna con su respectivo valor en la segunda. Durante este proceso además se leen los archivos de pesos, pero estos no tienen un impacto significativo en la complejidad temporal puesto que no se realizan recorridos en ellos.

```
public void leerArchivoAccidentesC(String rutaAccidentes, String rutaPeso, int numEntradas, String punto)
	{
		File f = new File(rutaAccidentes);
		File p = new File(rutaPeso);
		try {
			JSONTokener tkn = new JSONTokener(new FileReader(f));
			JSONArray arr = new JSONArray(tkn);
			JSONTokener tk = new JSONTokener(new FileReader(p));
			JSONArray ar = new JSONArray(tk);

			for(int i = 0; i < numEntradas; i++)
			{
				Double numerador = 0.0;
				Double denominador = 0.0;
				JSONObject peso = ar.getJSONObject(0);
				JSONObject actual = arr.getJSONObject(i);
				JSONArray info = actual.names();
				String localAuthority = actual.getString("Ciudad");
				String codigo = actual.getString("Code");


				for(int j = 0; j < info.length(); j++)
				{
					try{
						String txt = info.getString(j);
						int anho = Integer.parseInt(txt.substring(0,4));
						int cantidad = (int) actual.get(""+anho);
						Double pesoAnho = (double) peso.get(anho+"");
						numerador+=cantidad*pesoAnho;
						denominador+=pesoAnho;
					}
					catch(Exception e){
						continue;
					}
				}
				Double ps = numerador/denominador;
```

```
                                LocalAuthority la = new LocalAuthority(localAuthority, codigo,
ps);
                                if (punto.equals("A")){
                                        punto1A.add(la);
                                }
                                else if (punto.equals("B")){
                                        punto1B.add(la);
                                }
                        }
                }
                catch (FileNotFoundException e) {
                        e.printStackTrace();
                }
        }
```

La primera consulta tiene una complejidad de O(n) puesto que es lo que le toma al heap recorrer todos los datos e imprimirlos, ya que están organizados (lo cual toma log(n)).

```
public void consulta2A (String pCodigo, int pAnho)
        {
                TablaHashLP<Integer, Integer> local = tablaDatosA.get(pCodigo);
                int cantidad = local.get(pAnho);
                System.out.println("Codigo local authority: " + pCodigo + "\n" + "Ocurrieron " +
cantidad + " accidentes en el a?o " + pAnho + ".");
        }
```

La segunda consulta tiene una complejidad temporal constante puesto que los datos ya se encuentran en la tabla de Hash por lo cual solo es necesario realizar get dos veces, con lo cual por medio de la función de Hash se obtiene el valor buscado.

```
public void consulta3A (String pNombre, int pAnho)
        {
                String codigo = arbol3Anombres.get(pNombre);
                TablasFlujo tablas = arbol3AFlujo.get(codigo);
                TablaHashLP<Integer, Integer> todos = tablas.darTablaFlujoTodosA();
                TablaHashSC<Integer, Integer> carros = tablas.darTablaFlujoCarrosA();
                int cantAll = todos.get(pAnho);
                int cantCarros = carros.get(pAnho);
                consulta2A(codigo, pAnho);
                System.out.println("En el a?o " + pAnho + " transitaron " + cantCarros + "
carros y un total de " +
                                cantAll + " vehiculos.");
        }
```

Laura Maya

201423854

Carlos Peñaloza

201531973

La tercera consulta tiene una complejidad temporal de O(log(n)) puesto que se buscan valores en una tabla de Hash, lo cual toma tiempo constante, y en árboles rojo negro, lo cual toma log(n) al ser esta la altura máxima, y por lo tanto, la cantidad de elementos que se tendría que recorrer en el peor caso.