Reto 3: Documento de análisis

Integrantes del grupo

Grupo 11: **Federico Melo Barrero.**

202021525.

[f.melo@uniandes.edu.co](mailto:f.melo@uniandes.edu.co)

Implementa el requerimiento individual 2.

**Juan Camilo Prieto Avella.**

201814815.

[jc.prietoa@uniandes.edu.co](mailto:jc.prietoa@uniandes.edu.co)

Implementa el requerimiento individual 3.

Análisis de complejidad de los requerimientos

Requerimiento 1

Para el requerimiento 1 se utiliza un árbol rojo-negro creado en la función **create\_tree\_req1**. El árbol tiene como llaves ciudades y como valores árboles. Cada uno de los árboles que son valores tiene como llaves fechas (con hora, minuto y segundo, en formato **datetime.datetime**) y el valor correspondiente a cada llave es el avistamiento en la ciudad dada a la fecha y hora de la llave. Esto se puede hacer así porque en los datos del archivo **UFOS-utf8-large.csv** no hay dos avistamientos que hayan ocurrido en la misma ciudad y en la misma fecha, hora, minuto y segundo.

Haciendo uso del RBT, la función **requirement1** obtiene las respuestas del requisito 1 de la siguiente forma:

* Obtiene el número de ciudades con avistamientos de OVNIS sacando el tamaño del árbol de ciudades, lo que logra mediante la operación **om.size(tree\_req1)** y presenta una complejidad temporal de O(1).
* Obtiene la ciudad con más avistamientos de OVNIS sacando una lista de los valores y llaves del árbol para iterarlos y buscar cual es la ciudad que el tamaño de sus valores sea mayor y esto presenta una complejidad temporal de O(n).
* Obtiene el número de avistamientos de OVNIS en la ciudad dada sacando el tamaño del respectivo árbol de fechas de la ciudad, lo que logra mediante la operación **om.size(tree\_cities)** y presenta una complejidad temporal de O(1).
* Construye un arreglo con los tres primeros y tres últimos avistamientos, respetando el orden cronológico, realizando un **me.getValue()** en el árbol de fechas para las tres primeras y las tres últimas llaves. Cada **get** presenta una complejidad temporal de O(log n).

En el archivo **view.py**, se usa la función **print\_req1()** para mostrar los resultados del requerimiento uno en una tabla, hecha con la librería **prettytable**. Esta función no interviene en el funcionamiento del requerimiento.

Requerimiento 2

Implementado por Federico Melo Barrero.

Para el requerimiento 2 se emplea un RBT creado en la función **create\_tree\_req2**. El árbol tiene como llaves duraciones en segundos y como valores árboles. Cada árbol de los que son valores tiene como llaves cadenas de caracteres de la forma **'ciudad- zz'**, donde **zz** representa el acrónimo del país, y como valores arreglos que contienen todos los avistamientos de dicha duración en segundos que ocurrieron en el país y ciudad dados. Para los avistamientos que no tienen país se pone **'zz'** como acrónimo del país en el nombre de la llave, sin cambiar esto los datos internos del avistamiento (es decir, en la tabla de respuesta final no sale zz en el país, sale vacío. Las cadenas de caracteres para las llaves se construyen con la función **country\_city\_key**). Para realizar este requisito se tuvo en cuenta que en los datos ocurre que hay varios avistamientos con igual duración en segundos que ocurrieron en la misma ciudad.

Con base en el RBT, la función **requirement2** construye las respuestas del requisito 2 de la siguiente manera:

* Obtiene el número de duraciones distintas de avistamientos de UFO usando la operación **om.size(tree\_req2)** sobre el árbol que tiene como llaves duraciones en segundos. Eso presenta una complejidad temporal de O(1).
* Obtiene la duración en segundos más larga registrada usando la operación **om.maxKey(tree\_req2)** sobre el árbol que tiene como llaves duraciones en segundos. Eso presenta una complejidad temporal de O(log n).
* Obtiene el total de avistamientos registrados con duración máxima sacando el valor para la llave que es la mayor duración en segundos, que se obtuvo antes con **om.maxKey**. Se hace un **om.get()** que presenta complejidad temporal deO(log n). Luego, se obtiene el tamaño del valor, que es un RBT, con la operación **om.size()**, lo que tiene un costo en complejidad temporal de O(1). En total, la complejidad es de O(log n) + O(1) = O(log n +1) = O(log n).
* Obtiene el rango de duraciones en segundos entre las duraciones máxima y mínima dadas por el usuario usando un **om.values()** sobre el árbol que tiene duraciones en segundos como llave, lo cual presenta una complejidad temporal de O(log n). Luego, usa dos ciclos independientes sobre esos valores para obtener los 3 primeros y 3 últimos avistamientos del rango. Cada ciclo corre como máximo 3 veces (obsérvense las condiciones en los while). En el primer ciclo, **om.valueSet()** sobre el árbol de ciudades, lo cual tiene una complejidad temporal de O(log n).. Para obtener los últimos 3 avistamientos del rango se usa la misma idea: un ciclo que corre a lo más 3 veces y efectúa un **om.valueSet()** sobre el árbol que tiene longitudes como llave: cuesta O(log n). Es decir, en total, la complejidad temporal es de O(log n) + O(log n) + O(log n) = O(3 log n) = O(log n).
* Cuenta el número de avistamientos en el rango recorriendo cada valor de los valores del árbol de duraciones en segundos, que son árboles de ciudades, y obteniendo el tamaño de las listas en ellos. La complejidad temporal de esto depende del rango dado, y en el peor caso, en el que el rango son todos los elementos, la complejidad es de O(N).

En el archivo **view.py**, se usa la función **print\_req2()** para mostrar los resultados del requerimiento dos en una tabla, hecha con la librería **prettytable**. Esta función no interviene en el funcionamiento del requerimiento.

Requerimiento 3

Implementado por Juan Camilo Prieto Avella.

Para el requerimiento 3 se implementa un RBT con la función de **create\_tree\_req2**. El árbol tiene como llaves la hora del avistamiento en formato %H:%M:%S y como valores arreglos con los avistamientos por fecha.

Ahora usamos la función **requirement3** que necesita el RBT creado anteriormente para crear las siguientes respuestas:

* Obtiene el tamaño del árbol para saber todas las distintas horas que se tienen con una complejidad temporal de O(1).
* Obtiene la hora mas tarde en la que se a observado un OVNIS en la base de datos con la operación **om.maxKey(tree\_req3)**, que presenta una complejidad temporal de O(log n). Recupera el número de avistamientos que tuvieron lugar en esa hora haciendo un **om.get()** para esa llave, con una complejidad temporal de O(log n). Eso retorna un arreglo con los avistamientos de esa fecha, así que se saca su tamaño con **lt.size()**, que presenta una complejidad temporal de O(1). Además, la complejidad temporal total es de O(log n) + O(log n) + O(1) = O (2 log n +1 ) = O(log n).
* Obtiene el número total de avistamientos en el rango de horas usando la operación **om.values(tree\_req3, horaMin, horaMax)** sobre el árbol, lo cual tiene una complejidad de O(log n), para obtener una lista simplemente encadenada con los valores en el rango de fechas, es decir, una lista simplemente encadenada de arreglos. Para cada arreglo en la lista, se obtiene su tamaño con **lt.size(sightings\_array)**, cada uno con complejidad O(1), y se suman todos los tamaños para el número total de avistamientos. La complejidad de todos los **lt.size()** nunca es O(n) porque no se cuenta avistamiento por avistamiento, sino que se saca el tamaño de las listas y esos se suman. La complejidad acaba siendo entonces la del **om.values()**, que es O(log n), pero la complejidad en alguna prueba particular depende del tamaño del rango.
* Genera un arreglo con los tres primeros y tres últimos avistamientos dentro del rango en orden cronológico tomando los tres primeros y los tres últimos de la lista de valores que se obtuvo con **om.values(tree\_req3, horaMin, horaMax)**. El **om.values()** presenta una complejidad de O(log n), aunque este costo ya se tuvo en cuenta.

En el archivo **view.py**, se usa la función **print\_req3()** para mostrar los resultados del requerimiento tres en una tabla, hecha con la librería **prettytable**. Esta función no interviene en el funcionamiento del requerimiento, solo se usa para imprimir dichos resultados.

Requerimiento 4

Para el requerimiento 4 se hace uso de un árbol rojo-negro creado en la función **create\_tree\_req4**. El árbol tiene como llaves las fechas de avistamientos (únicamente las fechas, no las horas) y como valores arreglos con los avistamientos por fecha.

Utilizando el RBT, la función **requirement4** genera las respuestas del requisito 4 de la siguiente forma:

* Obtiene el número de fechas distintas en las que ocurrieron avistamientos de OVNIS con la operación **om.size(tree\_req4)** sobre el RBT, la cual presenta una complejidad temporal de O(1).
* Obtiene la fecha del avistamiento de OVNIS más antiguo de la base de datos con la operación **om.minKey(tree\_req4)**, que presenta una complejidad temporal de O(log n). Recupera el número de avistamientos que tuvieron lugar en ese día haciendo un **om.get()** para esa llave, con una complejidad temporal de O(log n). Eso retorna un arreglo con los avistamientos de esa fecha, así que se saca su tamaño con **lt.size()**, que presenta una complejidad temporal de O(1). Ergo, la complejidad temporal total es de O(log n) + O(log n) + O(1) = O (2 log n +1 ) = O(log n).
* Obtiene el número total de avistamientos en el rango de fechas usando la operación **om.values(tree\_req4, fechaMin, fechaMax)** sobre el árbol, lo cual tiene una complejidad de O(log n), para obtener una lista simplemente encadenada con los valores en el rango de fechas, es decir, una lista simplemente encadenada de arreglos. Para cada arreglo en la lista, se obtiene su tamaño con **lt.size(sightings\_array)**, cada uno con complejidad O(1), y se suman todos los tamaños para el número total de avistamientos. La complejidad de todos los **lt.size()** nunca es O(N) porque no se cuenta avistamiento por avistamiento, sino que se saca el tamaño de las listas y esos se suman. La complejidad acaba siendo entonces la del **om.values()**, que es O(log n), pero la complejidad en alguna prueba particular depende del tamaño del rango.
* Genera un arreglo con los tres primeros y tres últimos avistamientos dentro del rango en orden cronológico tomando los tres primeros y los tres últimos de la lista de valores que se obtuvo con **om.values(tree\_req4, fechaMin, fechaMax)**. El **om.values()** presenta una complejidad de O(log n), aunque este costo ya se tuvo en cuenta.

En el archivo **view.py**, se usa la función **print\_req4()** para mostrar los resultados del requerimiento cuatro en una tabla, hecha con la librería **prettytable**. Esta función no interviene en el funcionamiento del requerimiento.

Requerimiento 5

Para el requerimiento 5 se emplea un RBT creado en la función **create\_tree\_req5**. El árbol tiene como llaves la coordenada latitud (aproximada a dos cifras decimales) y como valores árboles. Cada árbol de los que son valores tiene como llaves coordenadas de longitud (aproximadas a dos cifras decimales) y como valores arreglos que contienen todos los avistamientos que tomaron lugar en la latitud y longitud dada. Al realizar este requisito se tuvo en cuenta que en los datos pasa que hay avistamientos que ocurrieron en exactamente la misma latitud y longitud. Nótese que este árbol es muy similar al árbol que se usó para el requisito 2: un árbol de árboles con arreglos.

Utilizando el RBT, la función **requirement5** genera las respuestas del requisito 5 de la siguiente forma:

* Obtiene el rango de latitudes entre la latitud mínima dada por el usuario y la máxima usando un **om.values()** sobre el árbol que tiene latitudes como llave, lo cual presenta una complejidad temporal de O(log n). Luego, usa dos ciclos independientes sobre esos valores para obtener los 5 primeros y 5 últimos avistamientos del rango. Cada ciclo corre como máximo 5 veces (obsérvense las condiciones en los while). En el primer ciclo, se realiza un **om.values()** sobre el árbol que tiene longitudes como llave, lo cual tiene una complejidad temporal de O(log n). Para obtener los últimos 5 avistamientos del rango se usa la misma idea: un ciclo que corre a lo más 5 veces y efectúa un **om.values()** sobre el árbol que tiene longitudes como llave: cuesta O(log n). Es decir, en total, la complejidad temporal es de O(log n) + O(log n) + O(log n) = O(3 log n) = O(log n).
* Cuenta el número de avistamientos en el rango recorriendo cada valor de los valores del árbol de latitudes, que son árboles de longitudes, y obteniendo el tamaño de las listas en ellos. La complejidad temporal de esto depende del rango dado, y en el peor caso, en el que el rango son todos los elementos, la complejidad es de O(N).

En el archivo **view.py**, se usa la función **print\_req5()** para mostrar los resultados del requerimiento cinco en una tabla, hecha con la librería **prettytable**. Esta función no interviene en el funcionamiento del requerimiento.

Requerimiento 6

Este requerimiento utiliza únicamente los resultados del requerimiento 5. Usa la librería **folium** para crear el mapa que se pide, el cual guarda como archivo HTML con nombre “**requerimiento6.html**” en el directorio Reto3-G11. Para visualizar el archivo bastante con abrirlo usando cualquier navegador.

En el archivo **view.py**, se usa la función **print\_req6()** para imprimir una cadena de caracteres que dice que se ha generado y guardado el archivo, e informa la cantidad de avistamientos que tuvieron lugar en el área dada por parámetro.