Documento de análisis del Reto 3

Estudiante 1- Maria Alejandra Moreno Bustillo Cod 202021603
Estudiante 2 - Juliana Delgadillo Cheyne Cod 202020986

I. Carga de datos

Para la carga de datos de este reto implementamos 6 secciones en el catálogo uno para cada requerimiento y el restante es una lista general que contiene todos los avistamientos y su información para poder usar como referencia para los otros árboles y mapas que construiremos. Para el primer requerimiento construimos una tabla de hash llamada "Sightings_per_city" en la cual las llaves eran los nombres de las ciudades y el valor asociado a estas son unos árboles de tipo RBT los cuales ordenaban los avistamientos de esas ciudades por fecha y tiempo en el que sucedieron. Para el requerimiento 2 se creó un árbol RBT en el cual las llaves eran duraciones, el valor asociado a estas es una lista de todos los avistamientos con dicha duración y al finalizar la carga de datos, cada una de estas listas se ordenaba por orden alfabético de la combinación ciudad-país. En el requerimiento 3 se realizó un mapa RBT con el nombre de "Sightings_per_time" en el cual se lleva a cabo una clasificación por hora donde las llaves son las horas registradas en el formato HH:MM (con segundo en 00 para todos los casos) y sus llaves asociadas corresponden a una lista donde se encuentran la afirmación de todos los avistamientos encontrados que ocurrieron en dicha hora. Adicionalmente, cada una de las listas creadas como valores de la llave, son organizadas tanto por fecha como por hora/minuto de menor a mayor. Para el requerimiento 4 se hizo uso de un mapa RBT llamado "Sightings_per_date" el cual clasificaba de manera ordenada los avistamientos por las fechas en las que sucedieron. El valor asociado a las llaves en cada nodo del árbol es una lista de los avistamientos que se dieron en esa fecha específica y al final de la carga, la lista asociada en cada nodo del árbol se le aplicaba un sort para poder ordenarlas internamente por hora y minutos. En el requerimiento 5 se construye un árbol de tipo RBT al que se le asigna como nombre "Sightings_per_location" en el cual las llaves son los valores de latitud aproximados con dos cifras significativas y sus respectivos valores son una lista (array_list) que contiene los avistamientos cuya latitud sea igual a la llave respectiva. La lista se encuentra organizada por los valores de latitud de menor a mayor por medio de la función comparelongitudes.

II. Requerimiento 1 – Grupal

En este primer requerimiento nos solicitaban el total de ciudades donde se reportaron avistamientos, los avistamientos en una ciudad específica y mostrarlos ordenados de manera cronológica por fecha y hora. En la función getCitySights se crea una lista auxiliar en la que se irán guardando los avistamientos encontrados en la ciudad analizada. Mediante la función get y getValue obtuvimos del mapa correspondiente el valor asociado a la ciudad que se buscaba. Este valor asociado es un mapa, por lo que, para poder encontrar todos los avistamientos, se recorrieron todas las llaves del árbol mediante un for y se utilizó otro for para poder ir concatenando los valores dentro de cada llave del árbol y agregarlos a la lista de respuesta que se entrega al final. Debido a que los valores en el árbol se encuentran ya ordenados por fecha y hora, al extraerlos y agregarlos a la lista de

respuesta, esta ya se encuentra ordenada. En cuanto a la complejidad final del algoritmo, sabemos que las búsquedas en los árboles tienen una complejidad de $O(\log_2(n))$ y adicionalmente los fors utilizados dan una complejidad de O(m*k), siendo m la cantidad de fechas con hora de avistamientos dentro de esa ciudad y k la cantidad de avistamientos de esa ciudad. Ambos valores normalmente no van a ser muy grandes y en el peor caso probablemente sean parecidos, puesto que como las fechas del árbol son muy específicas, sucederá que habrá muchos nodos del árbol con solo un avistamiento. La complejidad final del algoritmo es $O(\log_2(n) + O(m*k))$. Como tal, la complejidad final del algoritmo sería O(m*k) debido a que esta es mayor que $O(\log_2(n))$.

```
def getCitySights (analyzer, city):
    avistamientoslst = lt.newList('ARRAY_LIST', cmpfunction = cmphour)

    cityentry = mp.get(analyzer['Sightings_per_city'],city)
    city_dateindex = me.getValue(cityentry)
    city_date_keys = om.keySet(city_dateindex['DateSightsIndex'])

    for date in lt.iterator(city_date_keys):
        datetry = om.get(city_dateindex['DateSightsIndex'], date)
        date_value = me.getValue(datetry)

        for avistamiento in lt.iterator(date_value['Sightslst']):
            lt.addLast(avistamientoslst,avistamiento)

    return avistamientoslst
```

III. Requerimiento 2 – Individual (María Alejandra Moreno Bustillo)

En este segundo requerimiento nos pedían buscar aquellos avistamientos cuyas duraciones se encontrarán dentro de los limites definidos por el usuario. Se creó una lista auxiliar en la cual se irán almacenando los avistamientos que entren dentro del rango establecido. Mediante operaciones de los TAD Ordered maps se obtuvieron las consultas necesarias para el requerimiento; por un lado, para obtener el avistamiento de mayor duración de todos, se utilizó la función om.maxKey y con este se hizo uso de la función get y getValue para obtener dicho avistamiento. Por otro lado, se utilizó la función del TAD om.values para poder obtener los valores asociados a las llaves del árbol que se encuentren dentro del límite inferior y superior del rango solicitado. Con un for se recorre cada valor asociado a una llave perteneciente al rango y dentro de cada uno se utiliza un for que recorre esos valores y saca cada avistamiento dentro de este y lo agrega a la lista auxiliar durationlst. La complejidad final del requerimiento es de $O(\log_2(n)) + O(k*m)$ siendo k la cantidad de duraciones dentro del rango dado y m la cantidad de avistamientos dentro de cada una de esas duraciones (llaves). El $O(\log_2(n))$ se atribuye a la complejidad de las funciones de búsqueda dentro de los mapas ordenados cuando estos se encuentran debidamente balanceados, de lo cual estamos segura de que aplica para el árbol creado pues se usan árboles RBT. La complejidad temporal es muy buena, pues al final O(k*m) se podría ver como un O(n) siendo n la cantidad de avistamientos dentro del rango dado y esto se puede evidenciar en las pruebas pues las consultas se obtienen casi instantáneamente. De acuerdo a lo anterior, la complejidad final sería en este caso O(n).

```
def getDurationSights(analyzer,lim_inf, lim_sup):

    durationlst = lt.newList('ARRAY_LIST')

    duration_omap = analyzer['Sightings_per_duration']
    duration_max = om.maxKey(duration_omap)
    duration_max_entry = om.get(duration_omap,duration_max )
    duration_max_value = me.getValue(duration_max_entry)
    duration_max_size = lt.size(duration_max_value['Sightslst'])
    duration_rangevalues = om.values(duration_omap, lim_inf, lim_sup)

    for value in lt.iterator(duration_rangevalues):
        for avis in lt.iterator(value['Sightslst']):
            lt.addLast(durationlst,avis)
```

IV. Requerimiento 3 – Individual (Juliana Delgadillo Cheyne)

En el requerimiento 3 se desea conocer el avistamiento más tardío que se encuentre cargado en términos de hora y minuto, además se pretende mostrar los avistamientos ordenados cronológicamente por hora y fecha encontrados en un rango de tiempo el cual es ingresado por el usuario. En primer lugar, se realiza un formateo de las horas que ingresa el usuario como límite inferior y límite superior, utilizado la función .datetime, de la cual se extrae únicamente la hora, bajo el formato HH:MM: SS. A continuación, se crea una lista de tipo arreglo en cual se irán añadiendo los avistamientos presentes en el rango de tiempo (rangelst), cabe mencionar que por la forma en la que se realizó la carga de datos, los avistamientos entraran a esta lista ordenados por hora y por fecha. Dado que la información de los avistamientos clasificada por hora se encuentra en un árbol RBT, para acceder a la información almacenada, se utilizan operaciones de TAD Ordered maps. Para encontrar la hora del avistamiento más tardío se empleó om.maxKey, get, getValue y size. Donde la primera permite extraer la llave de mayor valor, get permite encontrar la pareja de la llave, getValue extrae el valor asociado, que en este caso corresponde a los avistamientos en la hora más tardía y size nos da el número de elementos presentes en dicha lista. A continuación, se pretende encontrar los avistamientos en un rango de tiempo, para ello se utiliza om.values, el cual extrae todos los valores del árbol que se encuentran entre la hora mínima y la máxima que se ingresa como parámetro. Luego, se crea un bucle for que itera con todos los valores asociados a las llaves que se encuentran dentro del rango y se realiza un for adicional que obtiene los avistamientos presentes en el valor y los agrega a la lista “rangelst”. La complejidad de este requerimiento está dada por $O(\log_2(n)) + O(k*m))$ donde $O(\log_2(n))$ corresponde a la complejidad de los comandos de búsqueda dentro del mapa ordenado (balanceado) y $O(k*m)$ se da por los bucles for anidados, donde k hace referencia a la cantidad de horas encontradas dentro del rango y m es la cantidad de avistamientos ocurridos en dichas horas. Si se toma en consideración el peor caso posible, es decir, que se deban guardar todos los avistamientos cargados, la complejidad seria de $O(n)$ un muy buen tiempo de respuesta ante el requerimiento.

```python
def getreq3(analyzer, lim_inf, lim_sup):

    date1 = datetime.datetime.strptime(lim_inf, '%H:%M:%S')
    date2 = datetime.datetime.strptime(lim_sup, '%H:%M:%S')

    hour1 = date1.time()
    hour2= date2.time()

    rangelst = lt.newList('ARRAY_LIST', cmpfunction = cmphour)

    time_omap = analyzer['Sightings_per_time']

    time_oldest = om.maxKey(time_omap)
    time_oldest_entry = om.get(time_omap,time_oldest)
    time_oldest_value = me.getValue(time_oldest_entry)
    time_oldest_size = lt.size(time_oldest_value['Sightslst'])

    time_inrange = om.values(time_omap,hour1,hour2)


    for time in lt.iterator(time_inrange):
        for avis in lt.iterator(time['Sightslst']):
            lt.addLast(rangelst,avis)


    return rangelst, time_oldest, time_oldest_size
```

V. Requerimiento 4 – Grupal

En el cuarto requerimiento se pide encontrar el número de avistamientos registrados con la fecha más antigua, al igual que, todos los avistamientos registrados dentro de un rango de fechas ingresado por el usuario en el formato AAA-MM-DD. En primer lugar, se lleva a cabo el formateo de los inputs utilizando la función .datetime y extrayendo únicamente la fecha. A continuación, se crea una lista de tipo arreglo llamada rangelst donde se guardarán todos los avistamientos encontrados en un rango de fechas, dada la organización realizada durante la carga de datos, dichos avistamientos ingresaran organizados cronológicamente. Como la información clasificada por fecha se encuentra en un árbol RBT llamado Sighting_per_time, se accede a él con operaciones TAD Ordered maps. Primero se usa minKey para acceder a la menor llave, es decir, la fecha más antigua registrada. Con esta información se realiza un get para encontrar la pareja de dicha llave y un getValue para extraer el valor asociado, el cual corresponde a los avistamientos para esa fecha, luego, sobre este valor se ejecuta la función size la cual devuelve el número de avistamientos encontrados. Posteriormente, se utiliza om.value para extraer del árbol todos los valores que se encuentran entre un rango de fechas especificado por el usuario. A continuación, se crean dos bucles for, uno contenido dentro del otro, donde el primero recorre los valores que se identificaron dentro del rango solicitado y el segundo itera sobre los avistamientos presentes en cada valor y los va ingresando uno por uno a la lista rangelst. Para este requerimiento se calculó una complejidad de $O((\log_2(n)) + O(k*m))$. Donde $O(\log_2(n))$ se da por los comandos de búsqueda utilizados dentro del mapa ordenado (balanceado) y O(k*m) corresponde a los dos bucles for, donde k es el número de fechas encontradas en el rango y m es la cantidad de avistamientos encontrados dentro de cada fecha. Si consideramos el peor caso posible, que sería tener que recorrer todos los avistamientos existentes, la complejidad sería O(n) una respuesta con muy buen tiempo ante el requerimiento.

```
def getSightsinRange(analyzer, lim_inf, lim_sup):

    lim_inf_f = (datetime.datetime.strptime(lim_inf,'%Y-%m-%d')).date()
    lim_sup_f = (datetime.datetime.strptime(lim_sup,'%Y-%m-%d')).date()
    rangelst = lt.newList('ARRAY_LIST')

    date_omap = analyzer['Sightings_per_date']
    date_oldest = om.minKey(date_omap)
    date_oldest_entry = om.get(date_omap,date_oldest)
    date_oldest_value = me.getValue(date_oldest_entry)
    date_oldest_size = lt.size(date_oldest_value['Sightslst'])
    date_inrange = om.values(date_omap,lim_inf_f,lim_sup_f)


    for date in lt.iterator(date_inrange):
        for avis in lt.iterator(date['Sightslst']):
            lt.addLast(rangelst,avis)

    return rangelst, date_oldest, date_oldest_size
```

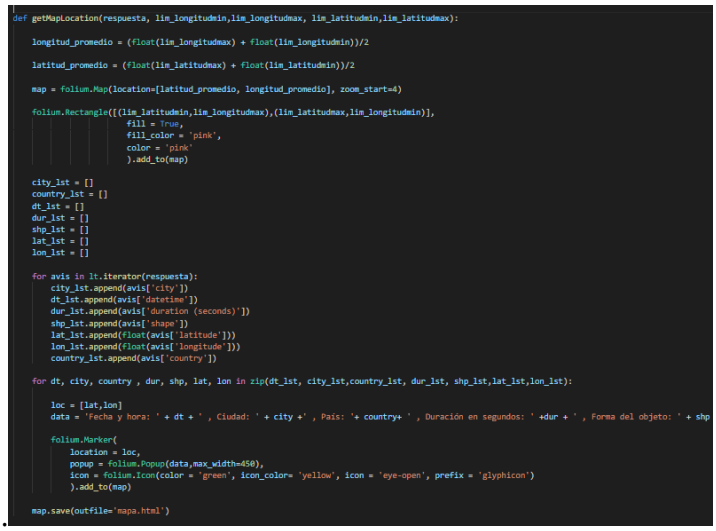
VI. Requerimiento 5 – Grupal

Para el requerimiento 5 se solicita encontrar el número de avistamientos cargados de una zona geográfica delimitada por un rango de longitudes y latitudes. Adicionalmente se pide mostrar información de algunos de los primeros y últimos cinco avistamientos encontrados. Para cumplir con este requerimiento, lo primero que se hace es crear un lista tipo Array_List, en la cual se almacenarán los avistamientos encontrados y se llama al árbol Sightings_per_location el cual contiene la información de los avistamientos clasificada por valores de latitud aproximados con dos cifras significativas. A continuación, se utiliza la operación om.values para extraer todos los valores que se encuentran en el rango ingresado por el usuario y se crea un for para iterar sobre estos. Dentro del ciclo, se extrae el valor de la longitud, el cual es un árbol que contiene los avistamientos de dicha latitud ordenados por longitud, y se realiza un om.values, esta vez para extraer todos los valores dentro del rango establecido con especto a la longitud. Luego, se crean dos bucles for, uno contenido dentro del otro, donde el primero recorre los valores que se identificaron dentro del rango solicitado de acuerdo a longitudes y el segundo itera sobre los avistamientos presentes en cada valor y los va gradando en la lista rangelst. Debido a que se utilizan tres ciclos dentro de la función se calcula una complejidad de $O((k*m*s) + (\log_2(n))$, donde k son los valores encontrados en un rango de acuerdo a latitud, m corresponde a todos los valores que se encuentran en el rango teniendo en cuenta la longitud, s son los avistamientos encontrados en la zona geográfica y $\log_2(n)$ está dado por la complejidad de los comandos de búsqueda que se utilizan al ingresar al mapa ordenado RBT. Si se considera el peor caso posible, lo que quiere decir que se deben recorrer todos los avistamientos cargados, la complejidad sería de O(n) para llegar a la solicitud del requerimiento.

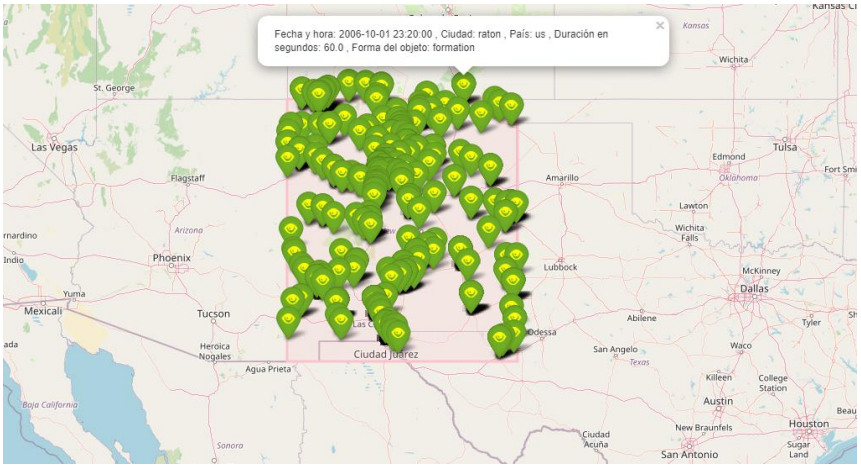
```
def getSightsLocation(analyzer, lim_longitudmin, lim_longitudmax, lim_latitudmin, lim_latitudmax):

    rangelst = lt.newList('ARRAY_LIST')

    latitude_tree = analyzer['Sightings_per_location']

    latitud_inrange = om.values(latitude_tree, lim_latitudmin, lim_latitudmax)

    for latitud in lt.iterator(latitud_inrange):

        longitude_tree = latitud['LongitudeTree']

        longitud_inrange = om.values(longitude_tree,lim_longitudmin, lim_longitudmax )

        for longitud in lt.iterator(longitud_inrange):
            for avis in lt.iterator(longitud['Sightslst']):
                lt.addLast(rangelst,avis)


    return rangelst
```

VII. Requerimiento 6 (Bono) – Grupal

El sexto requerimiento pedía visualizar los avistamientos es un mapa que se reportaron mediante el requerimiento 5 de acuerdo con latitud y longitud, lo cual quiere decir que los inputs que delimitan el rango son los mismos que se ingresan en el requerimiento anterior, por lo que se "reutiliza" la información y la respuesta del requerimiento 5. Para el lograr el requerimiento se importó una librería adicional: folium. Inicialmente, se calcula el promedio de la longitud y la latitud con respecto a los valores de límite inferior y superior ingresados por el usuario. Luego con esta información se ejecutan dos funciones de la librería folium; la primera folium.Map se utiliza para crear el mapa gráfico indicando la locación en términos de latitud y longitud que se desea abarcar, asimismo se incluye como parámetro zoom_start para determinar el acercamiento inicial del mapa. La segunda función es folium.Rectangle la cual se encarga de crear una figura en el mapa y define la forma en la que se encarrará el área solicitada (que en este caso será un rectángulo), el color de las líneas que delimitan dicha área y el color en el que se quiere representar la zona geográfica solicitada (para ambos casos el color elegido fue rosado). Posteriormente, se crean siente listas y por medio de un ciclo for que iteran los avistamientos pertenecientes a la zona geográfica delimitada, se le añade a cada una de estas listas los datos asociados a ellas, en los cuales se incluye ciudad, fecha/hora, duración, forma, latitud, longitud y país. A continuación, se crea un nuevo ciclo for en el que se recorren todas las listas creadas con la información relevante de cada avistamiento. Con la función folium.Marcker, se crean señaladores para los cuales se especifica posición dentro del mapa, colores, tamaño, diseño y el mensaje que se verá en pantalla cuando el usuario lo seleccione. Cabe mencionar que el mensaje desplegable es el que contiene la información básica del avistamiento. Finalmente, el mapa creado se guarda en un archivo .html, que se encuentra por fuera de las carpetas de la programación subida en github, por lo cual este solo podrá ser abierto por medio de un navegador. Para este requerimiento se calcula una complejidad de O (2n) en su peor caso, pues tendría que recorren todos los avistamientos una primera vez y luego recorrer la información básica de cada uno de ellos para la creación de los marcadores. Si se toma en cuenta el peor caso posible, en el que se debe pasar por todos los avistamientos cargados, se obtendría una complejidad final de O(n), para lograr cumplir con lo requerido.



```
def getMapLocation(respuesta, lim_longitudmin,lim_longitudmax, lim_latitudmin,lim_latitudmax):

    longitud_promedio = (float(lim_longitudmax) + float(lim_longitudmin))/2

    latitud_promedio = (float(lim_latitudmax) + float(lim_latitudmin))/2

    map = folium.Map(location=[latitud_promedio, longitud_promedio], zoom_start=4)

    folium.Rectangle([(lim_latitudmin,lim_longitudmax),(lim_latitudmax,lim_longitudmin)],
                     fill = True,
                     fill_color = 'pink',
                     color = 'pink'
                     ).add_to(map)

    city_lst = []
    country_lst = []
    dt_lst = []
    dur_lst = []
    shp_lst = []
    lat_lst = []
    lon_lst = []

    for avis in lt.iterator(respuesta):
        city_lst.append(avis['city'])
        dt_lst.append(avis['datetime'])
        dur_lst.append(avis['duration (seconds)'])
        shp_lst.append(avis['shape'])
        lat_lst.append(float(avis['latitude']))
        lon_lst.append(float(avis['longitude']))
        country_lst.append(avis['country'])

    for dt, city, country , dur, shp, lat, lon in zip(dt_lst, city_lst,country_lst, dur_lst, shp_lst,lat_lst,lon_lst):

        loc = [lat,lon]
        data = 'Fecha y hora: ' + dt + ' , Ciudad: ' + city +' , País: '+ country+ ' , Duración en segundos: ' +dur + ' , Forma del objeto: ' + shp

        folium.Marker(
            location = loc,
            popup = folium.Popup(data,max_width=450),
            icon = folium.Icon(color = 'green', icon_color= 'yellow', icon = 'eye-open', prefix = 'glyphicon')
            ).add_to(map)

    map.save(outfile='mapa.html')
```


Fecha y hora: 2006-10-01 23:20:00 , Ciudad: raton , País: us , Duración en segundos: 60.0 , Forma del objeto: formation
Kansas
Wichita
St. George
Las Vegas
Flagstaff
Amarillo
Edmond
Tulsa
Oklahoma
Lawton
Wichita Falls
Arizona
Phoenix
Lubbock
McKinney
Dallas
Yuma
Mexicali
Tucson
Abilene
Tyler
Odessa
San Angelo
Waco
Texas
Heroica Nogales
Agua Prieta
Ciudad Juárez
Killeen
College Station
Austin
Baja California
New Braunfels
Houston
Sugar Land
Ciudad Acuña
Sonora
San Antonio