Análisis de Reto 3 – Entrega Final

Integrantes:

* Juan David Vasquez Hernández - jd.vasquezh@uniandes.edu.co – 201914782
* Briseth Rodríguez Tovar – b.rodriguezt@uniandes.edu.co – 202116910

Complejidad de cada requerimiento:

Los códigos implementados para la solución de los requerimientos se muestran a continuación acompañados de sus respectivas complejidades.

En el caso de todos los algoritmos se usará **n** para representar el tamaño de UFOs y **k** para representar el número de llaves en el cual se organizan los datos en un árbol ordenado.

Req1:

def create\_city\_index(catalog):

    catalog['city\_index'] = om.newMap(omaptype='RBT', comparefunction=compareNames)

    city\_index = catalog['city\_index']

    for ufo\_data in lt.iterator(catalog['UFOs']):

        city\_info = ufo\_data['city']

        date\_info = ufo\_data['datetime']

        if om.contains(city\_index,city\_info):

            date\_index = om.get(city\_index,city\_info)['value']

            if om.contains(date\_index,date\_info):

                list\_UFOs = om.get(date\_index,date\_info)['value']

                lt.addLast(list\_UFOs, ufo\_data)

            else:

                list\_UFOs = lt.newList()

                lt.addLast(list\_UFOs,ufo\_data)

                om.put(date\_index,date\_info,list\_UFOs)

        else:

            date\_index = om.newMap(omaptype='RBT', comparefunction=compareDates)

            list\_UFOs = lt.newList()

            lt.addLast(list\_UFOs,ufo\_data)

            om.put(date\_index,date\_info,list\_UFOs)

            om.put(city\_index,city\_info,date\_index)

def getSightingsByCity(catalog, city):

    city\_index = catalog['city\_index']

    date\_index = om.get(city\_index,city)['value']

    return om.size(city\_index), date\_index

Para el primer requerimiento, la función create\_city\_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByCity( ) la complejidad es O(logk) en el caso en el que el árbol se encuentre balanceado. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k).

Req 2 (Briseth Rodríguez):

def create\_duration\_index(catalog):

    catalog['duration\_index'] = om.newMap(omaptype='RBT', comparefunction=compareDuration)

    duration\_index = catalog['duration\_index']

    for ufo\_data in lt.iterator(catalog['UFOs']):

        duration\_info = round(float(ufo\_data['duration (seconds)']),1)

        country\_info = ufo\_data['country']

        city\_info= ufo\_data['city']

        country\_city=  city\_info + '-' + country\_info

        if om.contains(duration\_index,duration\_info):

            country\_city\_index = om.get(duration\_index,duration\_info)['value']

            if om.contains(country\_city\_index,country\_city):

                list\_UFOs = om.get(country\_city\_index,country\_city)['value']

                lt.addLast(list\_UFOs, ufo\_data)

            else:

                list\_UFOs = lt.newList()

                lt.addLast(list\_UFOs,ufo\_data)

                om.put(country\_city\_index,country\_city,list\_UFOs)

        else:

            country\_city\_index = om.newMap(omaptype='RBT', comparefunction=compareLocation)

            list\_UFOs = lt.newList()

            lt.addLast(list\_UFOs,ufo\_data)

            om.put(country\_city\_index,country\_city,list\_UFOs)

            om.put(duration\_index,duration\_info,country\_city\_index)

def getSightingsByDuration(catalog,duration\_min,duration\_max):

    duration\_index = catalog['duration\_index']

    ufo\_list = lt.newList()

    latest\_duration = om.maxKey(duration\_index)

    latest\_country\_city = om.get(duration\_index,latest\_duration)['value']

    latest\_sightings = 0

    for country\_city in lt.iterator(om.keySet(latest\_country\_city)):

        latest\_sightings += lt.size(om.get(latest\_country\_city,country\_city)['value'])

    keys\_duration = om.keys(duration\_index,duration\_min,duration\_max)

    for key\_duration in lt.iterator(keys\_duration):

        country\_city\_index = om.get(duration\_index,key\_duration)['value']

        for key\_country\_city in lt.iterator(om.keySet(country\_city\_index)):

            ufo\_info = om.get(country\_city\_index,key\_country\_city)['value']

            for ufo in lt.iterator(ufo\_info):

                lt.addLast(ufo\_list,ufo)

    return latest\_duration, latest\_sightings, ufo\_list

Para el segundo requerimiento, la función create\_duration\_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByDuration( ) la complejidad es O(logk1 + logk1\*logk2) donde k1 son las llaves pertenecientes al árbol ordenado por duración (segundos) y k2 las llaves de cada sub árbol ordenado por la combinación ciudad-país – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk1 + 4\*logk1\*logk2) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk1 + 4\*logk1\*logk2). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de O(n) + O(k1 + k1\*k2)

Req 3 (Juan Vásquez):

def create\_time\_index(catalog):

    catalog['time\_index'] = om.newMap(omaptype='RBT', comparefunction=compareTime)

    time\_index = catalog['time\_index']

    for ufo\_data in lt.iterator(catalog['UFOs']):

        time\_info = ufo\_data['datetime'][11:19]

        date\_info = ufo\_data['datetime'][0:10]

        if om.contains(time\_index,time\_info):

            date\_index = om.get(time\_index,time\_info)['value']

            if om.contains(date\_index,date\_info):

                list\_UFOs = om.get(date\_index,date\_info)['value']

                lt.addLast(list\_UFOs, ufo\_data)

            else:

                list\_UFOs = lt.newList()

                lt.addLast(list\_UFOs,ufo\_data)

                om.put(date\_index,date\_info,list\_UFOs)

        else:

            date\_index = om.newMap(omaptype='RBT', comparefunction=compareTime)

            list\_UFOs = lt.newList()

            lt.addLast(list\_UFOs,ufo\_data)

            om.put(date\_index,date\_info,list\_UFOs)

            om.put(time\_index,time\_info,date\_index)

def getSightingsByTime(catalog,time\_min,time\_max):

    time\_index = catalog['time\_index']

    ufo\_list = lt.newList()

    latest\_time = om.maxKey(time\_index)

    latest\_dates = om.get(time\_index,latest\_time)['value']

    latest\_sightings = 0

    for date in lt.iterator(om.keySet(latest\_dates)):

        latest\_sightings += lt.size(om.get(latest\_dates,date))

    keys\_time = om.keys(time\_index,time\_min,time\_max)

    for key\_time in lt.iterator(keys\_time):

        date\_index = om.get(time\_index,key\_time)['value']

        for key\_date in lt.iterator(om.keySet(date\_index)):

            ufo\_info = om.get(date\_index,key\_date)['value']

            for ufo in lt.iterator(ufo\_info):

                lt.addLast(ufo\_list,ufo)

    return latest\_time, latest\_sightings, ufo\_list

Para el tercer requerimiento, la función create\_time\_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByTime( ) la complejidad es O(logk1 + logk1\*logk2) donde k1 son las llaves pertenecientes al árbol ordenado por tiempo HH:MM y k2 las llaves de cada sub árbol ordenado por fecha – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk1 + 4\*logk1\*logk2) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk1 + 4\*logk1\*logk2). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de O(n) + O(k1 + k1\*k2)

Req 4:

def create\_date\_index(catalog):

    catalog['date\_index'] = om.newMap(omaptype='RBT', comparefunction=compareDates)

    date\_index = catalog['date\_index']

    for ufo\_data in lt.iterator(catalog['UFOs']):

        date\_info = ufo\_data['datetime']

        if om.contains(date\_index,date\_info):

            list\_UFOs = om.get(date\_index,date\_info)['value']

            lt.addLast(list\_UFOs, ufo\_data)

        else:

            list\_UFOs = lt.newList()

            lt.addLast(list\_UFOs,ufo\_data)

            om.put(date\_index,date\_info,list\_UFOs)

def getSightingsByDate(catalog,initial\_date,final\_date):

    date\_index = catalog['date\_index']

    dates = lt.newList()

    keyMax = om.floor(date\_index,final\_date)

    final\_dates = om.get(date\_index,keyMax)['value']

    for date\_list in lt.iterator(om.values(date\_index,initial\_date,final\_date)):

        for date\_info in lt.iterator(date\_list):

            lt.addLast(dates,date\_info)

    return dates, final\_dates

Para el cuarto requerimiento, la función create\_date\_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByDate( ) la complejidad es O(logk) en el caso en el que el árbol se encuentre balanceado. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k).

Req 5:

def create\_coord\_index(catalog):

    catalog['coord\_index'] = om.newMap(omaptype='RBT', comparefunction=compareCoord)

    coord\_index = catalog['coord\_index']

    for ufo\_data in lt.iterator(catalog['UFOs']):

        latitude\_info = str(round(float(ufo\_data['latitude']),2))

        longitude\_info = str(round(float(ufo\_data['longitude']),2))

        if om.contains(coord\_index,latitude\_info):

            longitude\_index = om.get(coord\_index,latitude\_info)['value']

            if om.contains(longitude\_index,longitude\_info):

                list\_UFOs = om.get(longitude\_index,longitude\_info)['value']

                lt.addLast(list\_UFOs, ufo\_data)

            else:

                list\_UFOs = lt.newList()

                lt.addLast(list\_UFOs,ufo\_data)

                om.put(longitude\_index,longitude\_info,list\_UFOs)

        else:

            longitude\_index = om.newMap(omaptype='RBT', comparefunction=compareCoord)

            list\_UFOs = lt.newList()

            lt.addLast(list\_UFOs,ufo\_data)

            om.put(longitude\_index,longitude\_info,list\_UFOs)

            om.put(coord\_index,latitude\_info,longitude\_index)

def getSightingsByGeography(catalog,longitude\_min,longitude\_max,latitude\_min,latitude\_max):

    coord\_index = catalog['coord\_index']

    ufo\_list = lt.newList()

    keys\_latitude = om.keys(coord\_index,latitude\_min,latitude\_max)

    for key\_latitude in lt.iterator(keys\_latitude):

        latitude\_index = om.get(coord\_index,key\_latitude)['value']

        keys\_longitude = om.keys(latitude\_index,longitude\_min,longitude\_max)

        for key\_longitude in lt.iterator(keys\_longitude):

            ufo\_info = om.get(latitude\_index,key\_longitude)['value']

            for ufo in lt.iterator(ufo\_info):

                lt.addLast(ufo\_list,ufo)

    return ufo\_list

Para el quinto requerimiento, la función create\_time\_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByTime( ) la complejidad es O(logk1 + logk1\*logk2) donde k1 son las llaves pertenecientes al árbol ordenado por latitudes y k2 las llaves de cada sub árbol ordenado por longitudes – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk1 + 4\*logk1\*logk2) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk1 + 4\*logk1\*logk2). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de O(n) + O(k1\*k2)