RETO_003

Camilo Garcia – 201728914 – ca.garciar1@uniandes.edu.co REQ 2

Daniel Vargas – 201822068 - REQ 3

CARGA DE DATOS

```
44  def newAnalyzer():
45      """ Inicializa el analizador
46
47      Crea una lista vacia para guardar todos los crimenes
48      Se crean indices (Maps) por los siguientes criterios:
49      -Fechas
50
51      Retorna el analizador inicializado.
52      """
53      analyzer = {'avistamientos': None,
54                  'cityIndex': None
55                  }
56
57      analyzer['avistamientos'] = lt.newList('ARRAY_LIST', compareDates)
58      analyzer['cityIndex'] = om.newMap(omaptype='RBT',
59                                        comparefunction=compareCities)
60      analyzer['durationIndex'] = om.newMap(omaptype='RBT',
61                                        comparefunction=compareDurations)
62      analyzer['dateIndex'] = om.newMap(omaptype='RBT',
63                                        comparefunction=compareDates)
64      analyzer['latitudeIndex'] = om.newMap(omaptype='RBT',
65                                        comparefunction=compareCoordinate)
66
67      return analyzer
68
```

Para cargar los datos, se crea una lista tipo ARRAY_LIST, y 4 mapas siguiendo distintos criterios de organización para almacenar índices que apuntan a los datos de la lista. Para la creación de cada mapa, se utiliza el tipo RBT, ya que este minimiza los tiempos promedio y en peor escenario para la búsqueda de datos, aunque el proceso de carga sea algorítmicamente más complejo.

REQUISITO1: Consultar avistamientos en una ciudad

Para este requisito, se utilizan dos procedimientos. El primero busca la ciudad con más avistamientos, creando un nuevo mapa vacío que llena según el total de avistamientos en una ciudad.

```
233
234 def findMaxCity(analyzer):
235     cityValues = om.valueSet(analyzer)
236     orderCity = om.newMap(omaptype='RBT',
237                                         comparefunction=compareDates)
238     for city in lt.iterator(cityValues):
239         cont = city['count']
240         k= lt.firstElement(city['lstUFOs'])
241         om.put(orderCity, cont, k)
242
243     maxKey = om.maxKey(orderCity)
244     maxEntry = om.get(orderCity,maxKey)
245     maxValue = maxEntry['value']
246     maxCity = maxValue['city']
247
```

Aquí, si existen dos ciudades con el mismo número de avistamientos, ósea la llave, se reemplaza el valor, ya que solo nos interesa la ciudad con mayor numero de avistamientos. El orden de esta función, en el peor de los casos en el que cada avistamiento pertenezca a una ciudad distinta, sería O(n). Buscar el máximo tendría un orden de O(log(n)).

Para la segunda parte, se crea otro mapa que recibe únicamente los avistamientos que pertenecer a una ciudad de interés, y los ordenan según su fecha. Esto, para no tener que ordenar todos los avistamientos, sino únicamente los que nos interesan de la ciudad en específico.

```
296
297 def searchByCity(cont, city):
298     lstEntry = lt.newList('ARRAY_LIST')
299     cityInfo = om.get(cont, city)
300     cityValues = cityInfo['value']
301     cityOrdered = om.newMap(omaptype='RBT',
302                                         comparefunction=compareDates)
303     for event in lt.iterator(cityValues['lstUFOs']):
304         datetm= event['datetime']
305         entry = om.get(cityOrdered, datetm)
306         if entry is None:
307             datentry = newDataEntry(event)
308             om.put(cityOrdered, datetm, datentry)
309         else:
310             datentry = me.getValue(entry)
311             addCityIndex(datentry, event)
312
313     dates = om.valueSet(cityOrdered)
314     for dt in lt.iterator(dates):
315         for event2 in lt.iterator(dt['lstUFOs']):
316             lt.addLast(lstEntry,event2)
317
318     return cityValues['count'],lstEntry
319
```

Aquí, en el peor de los casos que todos los avistamientos estén incluidos en la ciudad de interés, habría una complejidad de O(nlog(n)), ya que por cada dato tendría que reordenar el mapa. Esta función retorna una lista para imprimir sus primeros y últimos 3 valores, que al ser ARRAY_LIST tiene un orden de O(1).

En el peor de los casos, tendría un orden de O(nlog(n)).

REQUISITO 2 (Camilo García): avistamientos por duración

Para este requisito, se utilizan dos procedimientos. El primero busca el avistamiento más largo, buscando el mayor elemento dentro del mapa de duraciones cargado al principio.

```
224
225 def findMax(analyzer):
226
227     maxKey = om.maxKey(analyzer)
228     maxEntry = om.get(analyzer,maxKey)
229     maxValue = maxEntry['value']
230     contador = maxValue['count']
231
232     return maxKey, contador
233
```

Esta operación tiene una complejidad de O(log(n)), en el caso de que cada avistamiento tenga una duración distinta, ya que el árbol balanceado optimiza los tiempos de búsqueda.

Para la segunda parte, se filtra el árbol de duraciones dentro del rango de búsqueda, y luego los eventos dentro del rango se agregan a un nuevo mapa que los ordena según la ciudad-pais. Esto con el propósito de solo ordenar los eventos que nos interesan dentro del rango. En el peor de los casos, si todos los eventos están dentro del rango de búsqueda, y cada uno tiene una duración distinta, tendría una complejidad O(n), ya que la creación de cada mapa tendría una complejidad de O(1).

```
265
266 def searchByDurationRange(cont, duration1, duration2):
267
268     values = om.values(cont, duration1, duration2)
269     counter = 0
270     lstEntry = lt.newList('ARRAY_LIST')
271     for lstvalues in lt.iterator(values):
272
273         counter += lstvalues['count']
274         mapByLocation = om.newMap(omaptype='RBT',
275                                   comparefunction=compareCities)
276
277         for event in lt.iterator(lstvalues['lstUFOs']):
278             country= event['country']
279             city= event['city']
280             countryCity= city +'-'+country
281             entry = om.get(mapByLocation, countryCity)
282             if entry is None:
283                 countryentry = newDataEntry(event)
284                 om.put(mapByLocation, countryCity, countryentry)
285             else:
286                 countryentry = me.getValue(entry)
287                 addCityIndex(countryentry, event)
288
289         countrycities = om.valueSet(mapByLocation)
290         for cou in lt.iterator(countrycities):
291             for event2 in lt.iterator(cou['lstUFOs']):
292                 lt.addLast(lstEntry,event2)
293
294     return counter, lstEntry
295
```

Este procedimiento tendría una complejidad de O(log(n)).

REQUISITO 3 (Daniel Vargas):

REQUISITO 4: avistamientos en un rango de fechas

Para este requisito, se hicieron dos procedimientos. Primero se busca dentro del mapa de dates el valor de la menor llave, para retornar el evento más reciente. Esto tiene una complejidad de O(log(n)), ya que utilizamos un árbol RBT que optimiza los tiempos de búsqueda al acortar las ramas.

```
208
209  def findMin(analyzer):
210
211      minKey = om.minKey(analyzer)
212      minEntry = om.get(analyzer,minKey)
213      minValue = minEntry['value']
214      contador = minValue['count']
215
216      return minKey, contador
217
```

Para la segunda parte, se filtran los datos según un rango de fechas, utilizando la función om.values(). En el peor de los casos, suponiendo que tenga que añadir todos los eventos, tendría una complejidad de O(n).

```
249
250  def searchByDateRange(cont, fecha1, fecha2):
251
252      datetm1 = datetime.datetime.strptime(fecha1, '%Y-%m-%d')
253      date1 = datetm1.date()
254      datetm2 = datetime.datetime.strptime(fecha2, '%Y-%m-%d')
255      date2 = datetm2.date()
256      values = om.values(cont, date1, date2)
257      counter = 0
258      lstEntry = lt.newList('ARRAY_LIST')
259      for lstvalues in lt.iterator(values):
260          counter += lstvalues['count']
261          for event in lt.iterator(lstvalues['lstUFOs']):
262              lt.addLast(lstEntry,event)
263
264      return counter, lstEntry
265
```

En el peor de los casos, tendría una complejidad de O(n).

REQUISITO 5: Avistamientos según zona geográfica

Para este requisito se utiliza una función que primero filtra los eventos que ocurrieron dentro de un rango de latitudes, luego crea un mapa nuevo, y ordena los eventos dentro de cada latitud por longitudes.

Posteriormente, vuelve a filtrar los datos por un rango de longitudes, y los añade a una lista.

La complejidad en el peor de los casos, que todos los elementos entren dentro del rango de longitudes y latitudes, y tengan longitudes y latitudes distintas, sería O(n), ya que la creación de cada mapa sería O(1) si se crea con solo un elemento.

```
319
320 def searchByLocation(cont, longitud_min, longitud_max, latitud_min, latitud_max):
321     longitud_max=float(longitud_max)
322     longitud_min=float(longitud_min)
323     latitud_min=float(latitud_min)
324     latitud_max=float(latitud_max)
325     values = om.values(cont, latitud_min, latitud_max)
326     counter = 0
327     lstEntry = lt.newList('ARRAY_LIST')
328     for lstvalues in lt.iterator(values):
329 
330         LongitudeOrd = om.newMap(omaptype='RBT',
331                                  comparefunction=compareCoordinate)
332 
333         for event in lt.iterator(lstvalues['lstUFOs']):
334             latitude_input= event['longitude']
335             latitude_1 = float(latitude_input).__round__(2)
336             entry = om.get(LongitudeOrd, latitude_1)
337             if entry is None:
338                 latitudeEntry = newDataEntry(event)
339                 om.put(LongitudeOrd, latitude_1, latitudeEntry)
340             else:
341                 latitudeEntry = me.getValue(entry)
342                 addCityIndex(latitudeEntry, event)
343 
344         double_filtered_values = om.values(LongitudeOrd, longitud_min, longitud_max)
345 
346         for lstvalues2 in lt.iterator(double_filtered_values):
347             for event2 in lt.iterator(lstvalues2['lstUFOs']):
348                 lt.addLast(lstEntry,event2)
349                 counter +=1
350 
351     return counter, lstEntry
```