Estructuras de Datos y Algoritmos: Reto 03

Juan Sebastián Ortega (Estudiante 1- js.ortergar1@uniandes.edu.co - 202021703) y Yesid Camilo Almanza (Estudiante 2 - y.almanza@uniandes.edu.co - 201921773)

Departamento de Ingeniería de Sistemas y Computación

UNIVERSIDAD DE LOS ANDES

10 de mayo de 2021

I. Introducción

CON base en la información presente en la base de datos *Context-Aware Music Recommender System* de Kaggle, específicamente en los archivos *context_content_featureslarge*, *user_track_hashtag_timestamp* y *sentiment_values*, se planteo realizar una aplicación de consola que permitiera cargar los datos contenidos en los archivos *.csv* y realizar un análisis de estos bajo ciertos requerimientos específicos. Con este objetivo, se implementó la librería *DISClib* para poder generar, filtrar y ordenar: listas, mapas de HASH y arboles binarios.

Requerimientos a cumplir:

1. Caracterizar reproducciones basándose en una característica de contenido y un rango determinado de valores.
2. Encontrar música para festejar (teniendo en cuenta las características de energy y danceability)
3. Encontrar música para estudiar (teniendo en cuenta las características de instrumentalness y tempo)
4. Estimar las reproducciones de los géneros musicales
5. Indicar el género musical más escuchado dentro de un tiempo en específico

En el momento de cumplir con los requerimientos se tuvo en mente la complejidad temporal que iba a adquirir el algoritmo en cada uno de estos, de forma que se tomaron las siguientes decisiones para optimizar la eficiencia:

- Todos los ordenamientos se realizarán mediante la aplicación de **MERGE SORTING** puesto que es el algoritmo de ordenamiento más efectivo dentro de la *DISClib* de acuerdo a pruebas de eficiencia realizadas previamente.
- Todas las listas serán instanciadas como **ARRAY_LIST** debido a que se prioriza la utilización de las funciones $lt.addLast()$ y $lt.getElement()$ las cuales cuentan con una complejidad de $O(K)$ en este tipo de estructura.
- Todos los mapas de Hash serán creados de tipo **PROBING** y se mantendrá el factor de carga óptimo por defecto que ya incluye la *DISClib*. Esto se hace con el objetivo de mantener una estructura simple que a su vez tenga la menor probabilidad de encontrar colisiones por lo que se busca reducir el tiempo gastado en la función *rehash*.
- Todos los árboles binarios serán arboles de tipo **Red-Black Tree**, esto se realiza con el objetivo de obtener un árbol balanceado que se acerque a la complejidad temporal de $O(log_2 N)$ para realizar cualquier operación.

II. Estructura de los datos cargados

La estructura de los datos cargados puede ser observada en el siguiente **apéndice.**

III. Funciones Auxiliares

El algoritmo de la aplicación fue diseñado de tal forma que los diferentes requerimientos comparten las mismas funciones (en su mayoría). Por lo tanto, antes de realizar el análisis general de la complejidad temporal de cada requerimiento es necesario analizar cada función principal individualmente. Las funciones que se mostrarán a continuación se encuentran ubicadas en el $model.py$ y son las encargadas de recorrer los arboles binarios, filtrar los mapas de hash y finalmente, organizar las listas.

En adición, existen ciertos tipos de registros específicos denotados de la siguientes forma:

- N: Nodos del árbol
- A: Artistas
- T: Tracks o eventos de escucha

III-A. Funciones de recorrido

- getTrackListByRange

```
def getTrackListByRange(analyzer, initialValue, finalValue, contentCharacteristic):
    lst = om.values(analyzer[contentCharacteristic], initialValue, finalValue)
    return lst
```

Análisis Temporal: Esta función tiene el objetivo de recibir el catalogo, dos valores que determinan un rango de búsqueda y una característica de contenido que indica la ubicación del árbol binario específico para esa consulta dentro del catalogo. De esta forma,

para lograr juntar todos los nodos dentro de este rango en una única lista (el retorno), se hace necesario acceder a dos nodos clave: el nodo identificado por el valor inicial y el nodo identificado por el valor final. Ya que se trata de un árbol RBT, la búsqueda de ambos elementos tendrá una complejidad temporal de $\mathbf{O(2log_2N)}$, la cual se puede simplificar a $O(log_2 N)$.

Análisis espacial: Ya que se retorna una lista con todos los nodos que entran dentro del rango establecido, el peor de los casos es obtener una lista con todos los nodos, es decir con un tamaño de $O(N)$.

- getTreeMapSize

```
def getTreeMapSize(track_list):

    tottracks = 0
    totartists = 0

    temporal_artist_map =  m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    for lstdate in lt.iterator(track_list):

        #Se buscan los artistas en una nueva tabla de hash para filtrar
        efectivamente los artistas unicos

        artist_lst =  m.keySet(lstdate["ArtistIndex"])

        for artist_id in lt.iterator(artist_lst):

            m.put(temporal_artist_map, artist_id, None)

        #Se obtiene el tamano (artistas unicos) del mapa temporal creado
        totartists = m.size(temporal_artist_map)

        tottracks += lt.size(lstdate['lsttracks'])

    return tottracks, totartists
```

Análisis Temporal: Esta función recibe una lista que incluye todos los nodos de un árbol binario en específico. Como se puede observar, se utiliza un **for loop** para iterar a través de todos los nodos (hash maps) que existen dentro del árbol y con esta iteración se realizan ciertas cuentas para obtener el total de artistas y pistas. Por lo tanto, la complejidad dependerá de la cantidad de nodos y de artistas dentro de estos, por lo que será: $O(A)$.

Análisis espacial: La función únicamente retorna dos número enteros cuyo espacio en memoria es totalmente despreciable. No obstante, para obtener dichos enteros se hace necesario crear un mapa de hash temporal que almacena el id de todos los artistas, por lo que el espacio ocupado por la ejecución de esta función dependerá del número de artistas únicos existentes y en el peor de los casos (cada artista sea único) tendrá una complejidad de: $O(A)$.

III-B. Funciones de filtro

- UniqueMap

```
def UniqueMap(lst):

    unique_map =  m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    #Se entra al arbol (ordenado por valor)
    for node in lt.iterator(lst):

        #Se entra a los valores del mapa (lista con tracks)
        track_lst =  node["lsttracks"]

        for track in lt.iterator(track_lst):
            #Se accede al track ID de cada track
            track_id = track["track_id"]

            #Se revisa si ese track_id ya es la llave del unique_map

            if not m.contains(unique_map, track_id):

                #En dado caso de que no lo contenga, a ndimos un track a esa llave
                m.put(unique_map, track_id, track)

    #De esta forma se retorna un diccionario cuyas llaves son los
    track_id y dentro de estas existe la informacion de un track unico

    return unique_map
```

Análisis temporal: Esta función recibe una lista que contiene todos los nodos de un árbol específico y con base en esta lista se obtiene un nuevo mapa de hash cuyas llaves son los track_id únicos. Nuevamente se utiliza un **for loop** para atravesar todos los nodos así como los tracks que estos contienen. Al tener que a travesar toda la información de la lista pasada como parámetro, la complejidad toma un valor de $O(T)$.

Análisis espacial: Ya que esta función guarda cada track_id único en un mapa de hash, en el peor de los casos, todo evento de escucha es completamente único por lo que sería necesario generar un diccionario que ocupa $O(T)$ en memoria, siendo T el número de eventos de escucha presentes en el catalogo.

- fusionMaps

```
def fusionMaps(map1, map2):
    #Fusiona mapas que tienen el mismo tipo de llave en un nuevo mapa  nico

    fusion_map =  m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    map1_keys = m.keySet(map1)

    for key in lt.iterator(map1_keys):

        if m.contains(map2, key):

            common_key = key

            entry = m.get(map2, key)

            common_value = me.getValue(entry)

            m.put(fusion_map, common_key, common_value)

    return fusion_map
```

Análisis temporal: Esta función toma dos mapas de hash y genera un mapa común con base en las llaves que ambos comparten. Para esto se hace necesario un **for loop** que recorre uno de los dos mapas ($O(N)$) y luego busca si las llaves de ese mapa están contenidas en el segundo mapa ($O(K)$). Por lo tanto, la complejidad total de esta función puede ser reducida a $O(N)$.

Análisis espacial:Ya que este mapa fusiona mapas por sus llaves comunes, el peor de los casos sería el caso en el que no exista ninguna llave común por lo que el tamaño del diccionario resultante sería igual a la suma del número de entradas en ambos diccionarios: $O(N_1 + N_2)$. Por practicad es posible reducir este valor a $O(N)$.

- randomSubListFromMap

```
def randomSubListFromMap(map, numelements):

    value_list = m.valueSet(map)
    map_size = m.size(map)

    random_list = lt.newList('ARRAY_LIST', compareIds)

    for n in range(numelements):
```

```
        random_pos = random.randint(1,map_size)
        random_item = lt.getElement(value_list, random_pos)
        lt.addLast(random_list, random_item)
    return random_list
```

Complejidad temporal: Esta función recibe un mapa y un número entero X para posteriormente generar una lista del contenido del mapa con X elementos. Ya que se hace necesario generar un valor aleatorio por cada número en el rango de (0, X) la complejidad temporal en el peor de los casos dependerá de X (que no puede ser mayor al número de entradas en el mapa), por lo tanto la complejidad es de $O(N)$.

Complejidad espacial: Ya que la lista puede llegar a tener la misma cantidad de elementos que el mapa ingresado, la complejidad espacial de esta función es de $O(N)$.

III-C. Funciones de ordenamiento

- listSort

```
def listSort(lst):
    sorted_list = merge.sort(lst,compareListItems)
    return sorted_list
```

Complejidad temporal: Esta función recibe una lista y utiliza el algoritmo de merge sort para organizarla. Debido a la complejidad propia de Merge Sort, la complejidad temporal toma el valor de $O(N \cdot log_2 N)$.

Complejidad espacial: En cuanto la complejidad espacial es bien sabido que merge sort necesita crear una lista adicional para llevar a cabo su proceso de organización por lo que sería necesario reservar un espacio aproximado de $O(2N)$.

IV. Requerimiento 01

```
def getReq1(analyzer, initialValue, finalValue, contentCharacteristic):
    """
    Retorna el numero de eventos de escucha en un rago de fechas.
    """
    node_list = getTrackListByRange(analyzer, initialValue, finalValue,
    contentCharacteristic)
    sizes = getTreeMapSize(node_list)
    return sizes
```

En el requerimiento 1 se aplican dos funciones principalmente: *getTrackListByRange* y *getTreeMapSize*. La primera es utilizada para encontrar los nodos que se encuentran dentro del rango introducido por el usuario mientras que la segunda obtiene información respecto al tamaño y contenido de dichos nodos. La sumatoria de ambas complejidades temporales puede ser vista como $O(A + log_2 N)$, donde A es el número de artistas únicos cargados y N es el número de nodos que tiene el árbol RBT.

En cuanto a la complejidad espacial esta sería una suma de primero, la lista con todos los nodos incluidos dentro del rango ($O(N)$) con el mapa generado para lograr contabilizar el tamaño de la información ($O(A)$) de forma que la complejidad temporal es de $O(N + A)$.

V. Requerimiento 02 - Estudiante 1

```
def getReq2(analyzer, energyMin, energyMax, danceMin, danceMax):
    """
    Retorna el numero de eventos de escucha en un rago de fechas.
    """
    node_list_energy = getTrackListByRange(analyzer, energyMin, energyMax, "energy")
    node_list_dance = getTrackListByRange(analyzer, danceMin, danceMax, "danceability")
    unique_energy = UniqueMap(node_list_energy)
    unique_dance = UniqueMap(node_list_dance)
    fusion_map = fusionMaps(unique_energy, unique_dance)
    #Gets the number of unique tracks
    fusion_map_size = m.size(fusion_map)
    #Gets 5 random tracks
    random_list = randomSubListFromMap(fusion_map, 5)
    return fusion_map_size, random_list
```

Para el Requerimiento 02, ya que se necesita filtrar los nodos de dos características diferentes, se hace necesario aplicar dos veces la función *getTrackListByRange* lo que acumularía una complejidad temporal de $O(2log_2 N)$. Posteriormente, sería necesario aplicar la función *UniqueMap* dos veces para obtener los tracks únicos en cada característica lo cual aumentaría en $O(2N)$ la complejidad temporal. Finalmente se aplicaría fusionMaps para encontrar los tracks en común entre ambas características (complejidad $O(N)$) y se obtiene una sublista al azar de estos tracks en común ($O(N)$). Al sumar toda la complejidad acumulada se obtiene: $O(2log_2 N + 4N)$

Por su parte, la complejidad espacial es bastante similar con la única diferencia de que la función *getTrackListByRange* ahora tiene una complejidad de $O(N)$. Por lo tanto, la complejidad espacial total sería de $O(6N)$.

VI. Requerimiento 03 - Estudiante 2

```
def getReq3(analyzer, instrumentalnessMin, instrumentalnessMax, tempoMin, tempoMax):
    """
    Retorna el numero de eventos de escucha en un rago de Instrumentalness and Tempo
    """
    node_list_instrumentalness=getTrackListByRange(analyzer,instrumentalnessMin,
    instrumentalnessMax, "instrumentalness")
    node_list_tempo=getTrackListByRange(analyzer,tempoMin, tempoMax, "tempo")
    unique_instrumentalness=UniqueMap(node_list_instrumentalness)
    unique_tempo=UniqueMap(node_list_tempo)
    fusion_map = fusionMaps(unique_instrumentalness, unique_tempo)
    #Gets the number of unique tracks
    fusion_map_size = m.size(fusion_map)
    #Gets 5 random tracks
    random_list = randomSubListFromMap(fusion_map, 5)
    return fusion_map_size, random_list
```

El Requerimiento 3 está construido de forma análoga al Requerimiento 2, por lo que su única diferencia son las dos características con las que se filtra la información dentro del catálogo. Por lo tanto, su complejidad tanto temporal como espacial es idéntica a la del requerimiento anterior. Complejidad temporal: $O(2log_2 N + 4N)$. Complejidad espacial: $O(6N)$.

VII. Requerimiento 04

```
def getReq4(analyzer, final_dict):
    tottracks_total = 0
    tottracks_map = m.newMap(numelements=30,
                             maptype='PROBING',
                             comparefunction=compareArtists)
    sizetracks_map = m.newMap(numelements=30,
```

```
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    uniqueartists_map = m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    for key in final_dict.keys():

        genre_name = key

        genre_min = final_dict[key]["min"]
        genre_max = final_dict[key]["max"]

        node_list_tempo = getTrackListByRange(analyzer, genre_min, genre_max,
        "tempo")

        sizes = getTreeMapSize(node_list_tempo)

        #Primer valor a mostrar -Total de eventos de escucha (por genero)
        tottracks_genre = sizes[0]


        #Total de artistas  nicos  por g nero
        uniqueartists_genre = sizes[1]
        m.put(uniqueartists_map, genre_name, uniqueartists_genre)


        m.put(tottracks_map, genre_name, lt.newList('ARRAY_LIST'))

        m.put(sizetracks_map, genre_name, tottracks_genre)


        #Se suma al total de los tracks para los g neros buscados
        tottracks_total += tottracks_genre

        #Se obtiene el ID de los primeros 10 artistas
        artist_count = 0

        for lstdate in lt.iterator(node_list_tempo):
            artist_lst = m.keySet(lstdate["ArtistIndex"])

            for artist_id in lt.iterator(artist_lst):

                entry = m.get(tottracks_map, genre_name)
                datentry = me.getValue(entry)

                lt.addLast(datentry, artist_id)
                artist_count += 1

                if artist_count >= 10:
                    break

            if artist_count >= 10:
                break


    #tottracks_map ---- Hash Map cuya llave es el g nero y cuyo valor una lista con
    #los 10 ids de los primeros artistas en aparecer

    #sizetracks_map ---- Hash Map cuya llave es el g nero y cuyo valor es la cantidad
    #de eventos de escucha por g nero

    #uniqueartists_map ----- N mero de artistas  nicos  para todo lo buscado

    #tottracks_total ----- Eventos de escucha totales

    return tottracks_total, sizetracks_map, uniqueartists_map, tottracks_map
```

Complejidad temporal: Esta función recibe tanto el catalogo con el árbol que contiene la información de toda la base de datos como un pequeño diccionario que incluye los valores de tempo máximo y mínimo referentes a cada género musical. A pesar de que existe un **for loop** que itera a través de todas las categorías, estas están definidas al rededor de una constante de 9 categorías por lo que le complejidad adjunta a estas será ignorada. En primera instancia se utilizan las funciones *getTrackListByRange* y *getTreeMapSize* para obtener los nodos dentro del rango de tempo establecido y obtener la información de su tamaño. Ambas operaciones acumulan $O(N + A)$. El resto del algoritmo se dedica a acceder a diferentes mapas de hash para ingresar cierta información, por lo que la complejidad de estos pasos siguiente es constante. Sin embargo, en la parte final del requerimiento se obtiene la información de 10 artistas por cada género buscado mediante un **for loop** que recorre todos los Artistas. Aunque esto le daría una complejidad adicional al requerimiento es posible observar que solamente se recorrerá este loop bajo una constante de 10 veces, por lo que se puede descartar su valor.

Complejidad espacial: En este requerimiento se busco fragmentar lo más posible la información que debía ser desplegada en consola por lo que se generaron múltiples mapas y variables de almacenamiento con este objetivo. Es posible observar la generación de **tottracks_map** que contiene un elemento por género, *uniqueartists_map* que funciona exacamente igual y *sizetracks_map* también. Por lo tanto, todos los mapas adicionales pueden ser descartados en el análisis espacial puesto que la información que guardan esta determinada tanto por la constante de géneros como de un máximo de 10 artistas. Por lo tanto, la complejidad espacial también se reduce a la de las funciones adicionales: $O(N+A)$

VIII. Requerimiento 05

```
def getReq5(analyzer, initialDate, finalDate, final_dict):

    tot_plays = 0

    genre_map = m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)

    genre_list = lt.newList("ARRAY_LIST")

    node_list_date = getTrackListByDate(analyzer,initialDate, finalDate, "created_at")

    unique_map =  m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)


    #Se entra al arbol (ordenado por valor)
    for node in lt.iterator(node_list_date):

        #Se entra a los valores del mapa (lista con tracks)
        track_lst =  node["lsttracks"]

        for track in lt.iterator(track_lst):
            #Se accede a cada track
            tot_plays += 1

            try:
                tempo = track["tempo"]
            except:
                #En dado caso de que no exista informaci n referente a tempo de un track,
                #se ignora y se pasa al siguiente

                continue

            for key in final_dict.keys():

                if float(tempo) >= final_dict[key]["min"]
                and float(tempo) <= final_dict[key]["max"]:

                    if m.contains(genre_map, key):

                        entry = m.get(genre_map, key)

                        datentry = me.getValue(entry)

                        datentry += 1

                        m.put(genre_map, key, datentry)

                    else:
                        m.put(genre_map, key, 0)



    for genre in lt.iterator(m.keySet(genre_map)):

        mini_list =  lt.newList("ARRAY_LIST")

        #Se a ade el g nero como elemento 1 a la mini lista
        lt.addLast(mini_list, genre)

        #Se a ade el valor del g nero como elemento 2 de la minilista
        entry = m.get(genre_map, genre)
        datentry = me.getValue(entry)
        lt.addLast(mini_list, datentry)

        #Se a ade la mini lista a la lista con todos los g neros (lista para ser organizada)
        lt.addLast(genre_list, mini_list)

        #Se organiza la lista de generos
        genre_list = listSort(genre_list, "ListItems")


    #Se accede al primer elemento de la lista (g nero con m s reproducciones)
    entry = lt.getElement(genre_list, 1)
    top_genre = lt.getElement(entry, 1)

    #Se itera sobre los tracks del primer g nero y se encuentran los tracks  nicos
    for node in lt.iterator(node_list_date):

        #Se entra a los valores del mapa (lista con tracks)
        hashtag_lst =  m.valueSet(node["HashtagIndex"])

        for hashtag in lt.iterator(hashtag_lst):

            track_list = hashtag["lsthashtags"]
```

```
    for track in lt.iterator(track_list):

        try:
            tempo = track["tempo"]
        except:
        #En dado caso de que no exista informaci n referente a
        #tempo de un track, se ignora y se pasa al siguiente

            continue

        track_id = track["track_id"]
        hash_value = track["hashtag"]

        if float(tempo) >= final_dict[top_genre]["min"]
        and float(tempo) <= final_dict[top_genre]["max"]:

            #Se revisa si ese track_id ya es la llave del unique_map
            if not m.contains(unique_map, track_id):

    #En dado caso de que no lo contenga, a adimos un track a esa llave
    #m.put(unique_map, track_id, lt.newList("ARRAY_LIST"))

    #Se a ade el hash_map como primer valor de la lista dentro del mapa
                hash_map = m.newMap(numelements=30,
                                    maptype='PROBING',
                                    comparefunction=compareArtists)
                entry = m.get(unique_map, track_id)
                datentry = me.getValue(entry)
                lt.addFirst(datentry, hash_map)



            entry = m.get(unique_map, track_id)
            datentry = me.getValue(entry)


            #Se accede al hash_map (primer elemento)
            hash_map = lt.getElement(datentry, 1)


            #Se revisa si el track con hash  nico  ya fue a adido
            #y si este tiene informaci n del vader

            if not m.contains(hash_map, hash_value):

                m.put(hash_map, hash_value, None)
                lt.addLast(datentry, track)

    top_unique_tracks = m.size(unique_map)


    #Ahora unique_map tiene como llave cada track  nico  y dentro una lista
    #con los eventos de
    #reproducci n que tienen hashtag distinto
    #Por lo tanto para encontrar el tama o de esta lista interna determinar
    #cu l tiene m s hashtags  nicos

    #Se pasa el unique_map a una lista organizable
    track_id_list = m.valueSet(unique_map)

    #Se organiza esa lista por el valor de el tama o de sus sub-listas
    track_id_list = listSort(track_id_list)

    #Se divide el track_id_list y se obtiene una lista con sus 10 primera canciones
    track_id_sublist = lt.subList(track_id_list, 1, 10)

#tot_plays ------- Total de reproducciones
#genre_list ------- Lista ORDENADA con los g neros y sus reproducciones
#top_genre ------- Nombre del g nero con m s reproducciones
#top_unique_tracks ------
#N mero de tracks  nicos  en el g nero con m s reproducciones
#track_id_sublist -------
#Lista con 10 elementos: Top 10 tracks con m s hashtags diferentes

    return tot_plays, genre_list, top_genre, top_unique_tracks, track_id_sublist
```

Complejidad Temporal: En este requerimiento se hace necesario filtrar múltiples veces el catalogo ingresado puesto que existe: un filtro por fecha, un filtro por eventos de escucha, un filtro por tracks únicos y un filtro que solo se aplica sobre los tracks pertenecientes a un género específico. Para lograr concretar dichos filtros tan específicos fue necesario recorrer todos los tracks existentes al menos dos veces lo que daría una complejidad total de $O(T)$.

Complejidad Espacial: En este requerimiento se utilizó el espacio para mantener registros de comparación utiles para solucionar ciertos aspectos. Debido a esto se utilizaron tres mapas de complejidad $O(N)$ cada uno así como una lista con esta misma complejidad.

IX. Pruebas físicas en la Máquina 1 y la Máquina 2

Con el objetivo de obtener datos reales respecto a la efectividad general de cada uno de los requerimientos, se planteo una prueba de rendimiento en dos máquinas diferentes. Estas pruebas replican los procedimientos de ejemplo mostrados en el documento de descripción del Reto 03, por lo cual se maneja la misma cantidad de información así como los mismos parámetros de búsqueda en ambas máquinas.

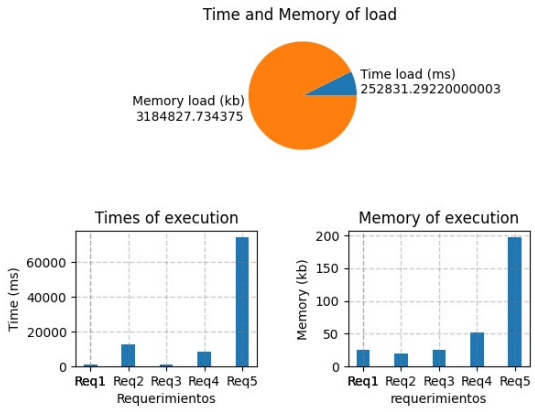

Figura 1. Rendimiento del Reto 03 en la Máquina 01

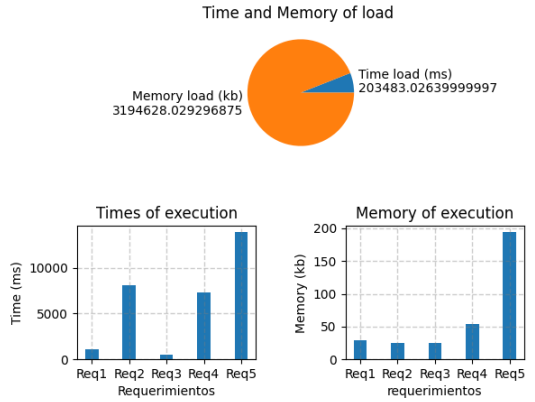

Figura 2. Rendimiento del Reto 03 en la Máquina 02

X. Conclusión

En conclusión, es posible observar en todos los requerimientos que los polinomios que modelan su complejidad son bastante similares entre si. Esto se debe a que la jerarquía de las estructuras de datos utilizada fue constante a través de todo el reto, por lo que las únicas diferencias eran ciertos parámetros de organización, pero siempre se seguía la estructura $Arbol ===> HashMap ====> lista$ que resulto bastante efectiva puesto que va de lo general a lo particular de forma ordenada.

El algoritmo creado fue implementado con éxito teniendo en cuenta la eficiencia temporal ya que ninguna función adquirió una complejidad aproximada mayor a $O(nlog_2 n)$, demostrando que es el ordenamiento el proceso más tardado en este tipo de aplicaciones que operan con grandes volúmenes de datos. En adición, otra ventaja que presenta el algoritmo presentado es que se prioriza primero la filtración de los datos y luego su ordenamiento. De esta forma, la cantidad

de datos a ordenar (que determina finalmente el tiempo de ejecución) es menor tras cada filtro. El orden seguido fue de vital importancia para ahorrar tiempo, puesto que los arboles rbt nos permitieron ordenar la información ya sea por fecha o valor desde la carga de datos, evitando la necesidad de utilizar algoritmos de ordenamiento en listas con gran cantidad de información.

REFERENCIAS

[1] R. Sedgewick and K. Wayne, Algorithms, 4th Edition. Addison-Wesley, 2011.

XI. APÉNDICES

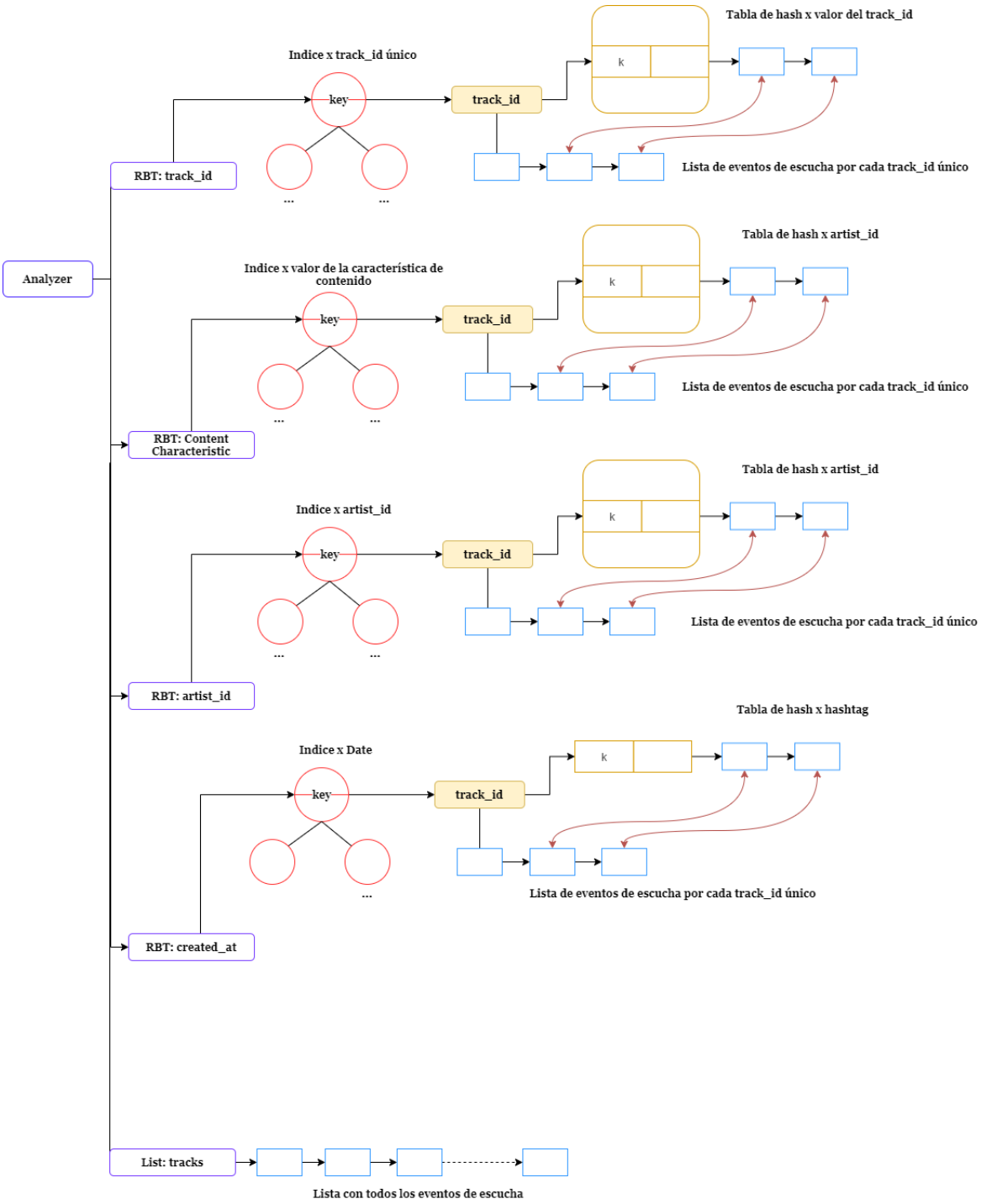

Figura 3. Caption