Análisis de complejidad Reto 3-G10

Juan Camilo Falla 201922219 - [j.fallag@uniandes.edu.co](mailto:j.fallag@uniandes.edu.co)

Nicolás Klopstock 202021352 – [n.klopstock@uniandes.edu.co](mailto:n.klopstock@uniandes.edu.co)

***Análisis de complejidad:***

1. *Requerimiento 1:*

El orden de complejidad de este requerimiento es *O(n\*m)*, siendo n el tamaño de la lista que devuelve la función “om.values()” y m, el tamaño de la lista de artistas únicos que se recorre para crear una tabla de Hash.

El requerimiento comienza con 3 *inputs* del usuario. Luego, se llama a la función “consultaArtistas()”, la cual entra en la tabla de Hash y busca la llave con el nombre de la característica ingresada por el usuario. Cuando encuentra la llave, devuelve el valor, que en este caso es un árbol, donde cada nodo es un valor de la característica, este proceso es de orden constante O(1). Entonces, se llama la función “om.values()” de la librería de DISC, la cual devuelve una lista de tamaño *n* con todos los valores de los nodos en un rango determinado. En nuestro caso, en cada índice de esa lista hay un diccionario en donde {‘reproducciones’: ‘ARRAY\_LIST’, ‘artistas’: ‘ARRAY\_LIST’, ‘canciones’: ‘ARRAY\_LIST’}. Se recorre esta lista y se entra a la llave “reproducciones” y “artistas”. Para el total de reproducciones, es simplemente entrar a la llave y devolver el size de la lista que es el valor de esa llave. Pero para los artistas es más complejo, porque un artista puede estar en la lista de diferentes valores de característica. Entonces, se recorre esta segunda lista de tamaño *m*. Cada valor se va guardando en una tabla de Hash (llave: Id del artista, valor: None), comparando que no se repitan los “ids” de los artistas. Con esta tabla, se usa la función de las librerías DISC “mp.keySet()”, esto crea una lista con las llaves de la tabla de Hash. De esta lista se saca el tamaño y eso se devuelve. Concluimos con que la función devuelve una tupla, donde [0] es el total de reproducciones de la característica y [1] es el total de artistas únicos.

Con la información de esta lista, se imprimen los resultados del requerimiento.

1. *Requerimiento 2 (Nicolas Klopstock):*

El orden de complejidad de este requerimiento es *O(n\*m)*, siendo *n* el tamaño de la lista que devuelve la función “om.values()” y *m*, el tamaño de la lista de canciones únicas.

El requerimiento comienza con 4 *inputs* del usuario. Luego, se llama a la función consultaReq2(). Esta función funciona con otra función dentro. Esta función interna es “consultaCanciones()”. Esta función es muy similar a la mencionada y usada en el requerimiento 1, pero en vez de devolver una tupla con el total de reproducciones y el total de artistas únicos, devuelve una tupla con el total de reproducciones y el total de canciones únicas. En un principio, se entra en la tabla de Hash y se busca la llave “energy”. Cuando encuentra la llave, devuelve el valor, que en este caso es un árbol donde cada nodo es un valor de la característica. Entonces, se llama la función “om.values()” de la librería de DISC, la cual devuelve una lista de tamaño n con todos los valores de los nodos en un rango determinado. En nuestro caso, en cada índice de esa lista hay un diccionario en donde {‘reproducciones’: ‘ARRAY\_LIST’, ‘artistas’: ‘ARRAY\_LIST’, ‘canciones’: ‘ARRAY\_LIST’}. Se recorre esta lista y se entra a la llave “reproducciones” y “canciones”. Para el total de reproducciones, es simplemente entrar a la llave y devolver el tamaño de la lista que es el valor de esa llave. Pero para las canciones es más complejo, porque una canción puede estar en la lista de diferentes valores de característica. Entonces, se recorre esta segunda lista de tamaño *m*. En esta lista, cada índice es un diccionario, donde {‘track\_id’: id de la canción, ‘instrumentalness’: valor de instrumentalness de la canción, ‘tempo’: valor de tempo de la canción, ‘danceability’: valor de danceability de la canción, ‘energy’: valor de energy de la canción}. Esta lista mencionada se recorre y cada elemento se guarda en otra lista.

La función que se mencionó al comienzo de esta explicación (consultaReq2()) usa esta función para juntar las canciones únicas de una característica con las únicas canciones de otra característica (en nuestro caso, Energy y Danceability), teniendo en cuenta, claro, que no se repitan entre sí. Esta función devuelve una lista con todas las canciones de las dos características sin repetición.

Con la información de esta lista, se imprimen los resultados del requerimiento.

1. *Requerimiento 3 (Juan Camilo Falla):*

El orden de complejidad de este requerimiento es *O(n\*m)*, siendo *n* el tamaño de la lista que devuelve la función “om.values()” y *m*, el tamaño de la lista de canciones únicas.

El requerimiento funciona exactamente igual al requerimiento anterior, pero con la diferencia de que no se usan las características “Energy” y “Danceability”, sino “Instrumentalness” y “Tempo”.

El requerimiento comienza con 4 *inputs* del usuario. Luego, se llama a la función consultaReq2(). Esta función funciona con otra función dentro. Esta función interna es “consultaCanciones()”. Esta función es muy similar a la mencionada y usada en el Requerimiento 1, pero, envés de devolver una tupla con el total de reproducciones y el total de artistas únicos, devuelve una tupla con el total de reproducciones y el total de canciones únicas. Explicada, se entra en la tabla de Hash y se busca la llave con el nombre de la característica ingresada por el usuario. Cuando encuentra la llave, devuelve el valor, que en este caso es un árbol donde cada nodo es un valor de la característica. Entonces, se llama la función “om.values()” de la librería de DISC, la cual devuelve una lista de tamaño n con todos los valores de los nodos en un rango determinado. En nuestro caso, en cada índice de esa lista hay un diccionario en donde {‘reproducciones’: ‘ARRAY\_LIST’, ‘artistas’: ‘ARRAY\_LIST’, ‘canciones’: ‘ARRAY\_LIST’}. Se recorre esta lista y se entra a la llave “reproducciones” y “canciones”. Para el total de reproducciones, es simplemente entrar a la llave y devolver el tamaño de la lista que es el valor de esa llave. Es importante considerar que existen canciones que pueden repetirse dentro de las dos características consultadas, es por esto por lo que se recorre esta segunda lista de tamaño *m*. En esta lista, cada índice es un diccionario, donde {‘track\_id’: id de la canción, ‘instrumentalness’: valor de instrumentalness de la canción, ‘tempo’: valor de tempo de la canción, ‘danceability’: valor de danceability de la canción, ‘energy’: valor de energy de la canción}. Esta lista mencionada se recorre y cada elemento se guarda en una lista aparte.

La función que se mencionó al comienzo de esta explicación (consultaReq2()) usa esta función para juntar las canciones únicas de una característica con las únicas canciones de otra característica (en nuestro caso, Instrumentalness y Tempo), teniendo en cuenta que solo existan los valores únicos de los “ids” de las canciones. Esta función devuelve una lista con todas las canciones de las dos características sin repetición.

Con la información de esta lista, se imprimen los resultados del requerimiento.

1. *Requerimiento 4:*

El orden de complejidad de este requerimiento es *O(n\*m)* cuando el usuario ingresa un género que no conoce el sistema donde *n* es la cantidad de nodos en el rango del tempo y *m* es la cantidad de artistas entre todos los nodos.

En caso de que el usuario desee conocer un género ya establecido en el sistema, el orden es O(*k*) donde *k* es la cantidad de artistas únicos en un género.

El requerimiento comienza con una serie de inputs requeridos que deben ser ingresados por el usuario para que se ejecute el programa. Primero el usuario debe ingresar cuantos géneros musicales desea consultar (máximo nueve géneros preestablecidos y si se desea, el usuario puede crear un género desconocido por el sistema, pero debe ingresar el nombre de este género y el rango del tempo. Cada vez que el usuario desea conocer un género se le pregunta si desea consultar uno de los géneros preestablecidos o si él desea ingresar el género.

En el primer caso, en el cual el usuario consulta un género predeterminado, se debe ingresar el nombre del género a conocer. A partir de esta información se consulta la tabla de hash en donde las llaves son los nombres de los nueve géneros que ya se conocen en el reto. Esta consulta es relativamente y tiene un orden de complejidad constante O(1). El valor asociado a cada género musical es otra tabla de hash en donde las llaves son los id de los artistas que tienen una canción asociada a ese género. Una vez se accede a esta tabla, se extrae un Keyset del mapa de los artistas, proceso el cual tiene una complejidad de O(*k*), donde *k* es la cantidad de llaves en la tabla. Una vez se obtiene la lista de los artistas únicos asociados a este género, se imprimen los resultados del requerimiento

En caso de que el usuario ingrese su género, se utiliza otro proceso. Una vez el programa tenga el nombre y el rango de tempo en el cual se encuentra este nuevo género, se utiliza la función “consulta artistas”. Esta acc a la tabla de hash principal y busca el valor de la llave “tempo” (proceso cuyo orden es O(1)). Una vez obtenido el valor asociado a esta llave, (un árbol en donde las llaves son los valores del tempo) se utiliza la función “om.values()” para recibir una lista con todos los nodos que se encuentran dentro del rango que ingresó por parámetro (el tamaño de esta lista es de *n* nodos). Por como organizamos la estructura, en cada índice de esa lista hay un diccionario en donde {‘reproducciones’: ‘ARRAY\_LIST’, ‘artistas’: ‘ARRAY\_LIST’, ‘canciones’: ‘ARRAY\_LIST’}. Esta lista se recorre para encontrar la cantidad de reproducciones y la cantidad de autores únicos. En cada nodo se suma el valor del tamaño de la lista de las canciones para encontrar el total de reproducciones en el rango esperado. Adicionalmente se recorren las listas de los artistas y cada artista es enviado a una tabla de hash temporal, en la cual el string de identificación del artista es la llave y el valor es None (con el fin de minimizar la cantidad de espacio adicional utilizado). Este proceso se repite por cada nodo del rango, lo cual significa que su orden de complejidad es de *O(n\*m)*, en el cual n es la cantidad de nodos dentro del rango y *m* es la cantidad artistas únicos en la lista dentro de cada nodo*.* Una vez se obtiene el nuevo mapa, se utiliza la función Keyset() y se con el tamaño de la lista retornada por esta última función se encuentra la cantidad de artistas únicos en el rango de tempo establecido por el usuario.

Con la información de esta lista, se imprimen los resultados del requerimiento.

1. *Requerimiento 5:*

El orden de complejidad de este requerimiento es *O(n\*m)*, siendo n el tamaño de la lista que devuelve la función om.values() y m, el tamaño de la lista de géneros con sus respectivas reproducciones y canciones con hashtags.

Como para este requerimiento se tuvo que hacer una nueva carga de datos, primero se va a explicar esta. La estructura es la siguiente: se crea un árbol donde cada nodo es una hora. El valor de cada nodo es un diccionario con dos llaves. Una es ‘fechaRepr’ (donde se guarda únicamente el entero del total de reproducciones en esa hora específica) y la otra es ‘generos’. En esta se guarda una tabla de Hash, en la que cada llave es el nombre de un género (‘metal’, ‘funk’, ‘r&b’, ‘chill-out’, etc.) y el valor de cada una es un diccionario segundo. Este tiene dos llaves. La primera es ‘reproducciones’, donde se guarda el entero del total de reproducciones de ese género en esa hora específica. La segunda es ‘canciones’, la cual tiene como valor una tabla de Hash donde cada llave es el id de una canción reproducida en la hora específica. El valor de cada una de estas en una lista de tipo ‘array’, la cual tiene todos los hashtags asociados a esa canción en esa hora específica. Aparte de esta estructura se tiene una tabla de Hash para relacionar cada hashtag con su vader. Esta tiene como llaves todos los hashtags y como valor, el vader asociado.

El requerimiento se dividió en dos partes para una mejor comprensión. La primera es en la que se imprime el total de reproducciones en el rango ingresado e imprime el top de géneros con sus respectivas reproducciones. La segunda parte es en el que se imprime el total de reproducciones, el total de canciones únicas y 10 canciones aleatorias del género con más reproducciones.

El requerimiento comienza con varios inputs en los que se pide el rango de horas de las que se quiere saber información. La primera parte comienza encontrando las reproducciones totales. Este proceso comienza en la función consultaGenero(). Por la manera en la que estamos cargando los datos, el algoritmo es usar la función om.values() para que devuelva una lista con los valores de las horas ingresadas en el árbol. Esta se recorre y se entra al diccionario de llave ‘fechaRepr’, el cual tiene un entero con el total de reproducciones de cada hora. En cada iteración se va sumando una en una y se devuelve el total cuando termina el recorrido. Es importante aclarar que en una función llamada mapaTempGen(), se hace un mapa donde las llaves son cada género y el valor es 0. Esto para una mejor organización para que luego quede claro el entero del total de reproducciones por cada género. Se debe recorrer la lista que devuelve om.values() y entrar al diccionario primero a la llave ‘géneros’, la cual tiene como valor una tabla de Hash. Se debe recorrer esa tabla y por cada género (por la manera en la que cargamos) se accede a la llave ‘reproducciones’ del diccionario segundo, la cual tiene como valor el entero del total de reproducciones de ese género en una hora específica. Entonces, se coge este entero y se agrega a la tabla de Hash anteriormente explicada (la de “mapaTempGen()”) y se guarda como valor de la llave con el nombre del género correspondiente. A este valor se le suma el número de reproducciones del género correspondiente cada iteración, es decir, por cada hora del rango ingresado. La función retorna el mapa de “mapaTempGen()” con el total de reproducciones de cada género.

Con el fin de tener un formato simple y útil para el print del requerimiento, en la función “consultaTopGeneros()” se crea una lista de tipo ‘array’. Luego, se necesita recorrer el mapa que retornó la función “consultaGenero()” (la explicada en el párrafo anterior) y se usa “mp.get()” para tener cada pareja del mapa. Cada pareja se agrega a la lista mencionada anteriormente y se retorna. A esta última pasa por un algoritmo de ordenamiento de tipo Quick, para saber cuál es el género con más reproducciones y el top.

Con la información de esta lista, se imprimen los resultados del requerimiento.

Para la parte dos, primero se usa la función “om.values()”. Esta te devuelve una lista con todos los valores asociados al rango de horas ingresado. Lo que sigue es entrar al mapa de los géneros y buscar la llave del género con más reproducciones. Se entra a ese mapa y se busca el mapa de sus canciones. Se recorre ese mapa, recorriendo también las listas asociadas a cada llave (sus hashtags). Se tienen que también recorrer las listas mencionadas. La idea de todo esto es crear y rellenar una nueva tabla de Hash, donde quedan como llave todas las canciones en ese rango de horas; sus valores, siendo estos una lista con todos los hashtags asociados a cada canción en el rango de horas. Con este mapa hecho, lo siguiente es la función “vaderPorUnaCancion()”. Lo que hace esta función es calcular el vader promedio de una canción dependiendo de sus vaders. Esta función es solo para una canción. Uno de los parámetros de la función es el id de la canción. Este id se busca en la tabla de Hash que se creó anteriormente y se devuelve el valor (la lista de hashtags). Esta lista se recorre y cada elemento se busca en la tabla de Hash aparte que se mencionó en un párrafo anterior (llave - hashtag, valor - vader). Se compara y se guarda el vader. Este se va sumando para tener su total y se divide entre el total de hashtags para sacar el promedio. La función retorna una tupla de 3 elementos: tupla[0] = entero del total de hashtags, tupla[1] = vader promedio y tupla[2] = entero del total de canciones únicos. También es importante mencionar que en el mismo ciclo donde se recorren las listas de hashtags, cada elemento se mete en una tabla de Hash nueva para confirmar que ninguno se repita. Este algoritmo se tiene que hacer para cada canción en la tabla de Hash total del rango de horas; eso hace “vaderPromedioParaCadaCancion()”. Esta función también crea una nueva tabla de Hash donde cada llave es el id de una canción y el valor es la tupla mencionada anteriormente.

Por último, para un sencillo formato para el print del requerimiento, en la función “topCancionesPorGenero()” se crea una lista de tipo ‘array’. Luego, se necesita recorrer el mapa que retornó la función vaderPromedioParaCadaCanción() y se usa mp.get() para tener cada pareja del mapa. Cada pareja se agrega a la lista mencionada anteriormente y se retorna. A esta última se le hace un ordenamiento de tipo “Merge”, para saber cuál es la canción con más hashtags.

***Análisis y Conclusiones:***

Haciendo el reto, pudimos aplicar y comprender de una mejor manera diferentes funciones y algoritmos muy relevantes cuando se usan árboles binarios, tablas de Hash y listas. También, pudimos comprobar, con datos de tiempo y de uso de memoria, cuál es la mejor manera de resolver un problema.

En esta sección se describirá la estructura que se usó y la explicación y análisis de porqué se eligieron las estructuras que se usaron. Luego, se analizarán los diferentes algoritmos usados para la búsqueda y las decisiones que se tomaron al momento de resolver el problema.

1. ***Estructura:***

La estructura elegida para este reto consta de una tabla de hash, en la cual las llaves son las características de las canciones y los valores son arboles donde las llaves de cada nodo en el árbol son los valores de dichas características.

Cada nodo del árbol es un valor de la característica de la cual el árbol es valor. Cada nodo tiene su valor, este siendo un diccionario de la manera: {‘reproducciones’: ‘ARRAY\_LIST’, ‘artistas’: ‘ARRAY\_LIST’, ‘canciones’: ‘ARRAY\_LIST’}. Como se puede ver, cada llave tiene como valor asociado una lista de tipo ‘array’. Para ‘reproducciones’ y para ‘artistas’, la lista es simplemente una lista en donde cada índice es un el id de cada canción (para ‘reproducciones’) y el id de cada artista sin repeticiones (para ‘artistas’).

Adicionalmente, existe una tabla de hash donde las llaves son los géneros que ya se encuentran predeterminados en el documento del reto 3 y los valores son un diccionario con la cantidad de reproducciones de dicho género, bajo la llave “reproducciones”. El otro valor de este diccionario se encuentra bajo la llave “artistas” y es una tabla de hash donde las llaves son los artistas de cada género y el valor es None, debido a que solo se necesita la cantidad de artistas únicos para el requerimiento 4.

El uso de diccionarios fue un poco arriesgado, ya que no pertenecen a los vistos en clase. El razonamiento que se tuvo fue, si se necesita acceder a información que debe estar clasificada según qué tipo de información es y no las llaves siempre son constantes, preferimos usar un diccionario (que, a la final, es una tabla de Hash). Se necesitaban solo 3 llaves y pensamos que crear una tabla de Hash únicamente para tan pocas llaves podía ser ineficiente en temas de uso de memoria, ya que usar un diccionario usa menos memoria que crear y llenar una tabla de Hash. Este mismo razonamiento pasó al uso de diccionarios dentro de la lista en la llave ‘canciones’ del diccionario primero. Se necesitaba clasificar información en diferentes tipos, pero la cantidad de llaves era bastante pequeña y eran pocos tipos. Entonces, por temas de uso de memoria, se decidió no crear y llenar una tabla de Hash, sino usar un diccionario.

Ya dentro del diccionario, cada llave tiene un valor, este siendo una lista de tipo ‘array’. Se decidió usar arreglos, ya que, según las pruebas hechas en el laboratorio 5 del curso, las listas de tipo ‘array’ son mucho más rápidas cuando se quiere agregar elementos consecutivos (ordenados) y se quieren recorrer (que es justo lo que hacemos en el reto). Y en caso de que fuese necesario aplicar un algoritmo de ordenamiento, el tiempo de respuesta al usar arreglos era significativamente menor que al usar listas enlazadas.

Teniendo en cuenta la información que recibe el sistema del usuario y la información que se debe retornar, se decidió utilizar árboles para las consultas y procesos en los primeros tres requerimientos y en el último. Esto se debe a que dicha estructura permite que se dé una búsqueda de manera efectiva de la información que se encuentre dentro del rango que desea conocer el usuario. Para los árboles podíamos escoger entre los árboles binarios de búsqueda, estructura que no se encuentra balanceada, o los árboles rojo y negro, los cuales si se encuentran balanceados. Con el fin de determinar cuál tipo se iba a usar, se realizaron una serie de pruebas cuyos resultados son los siguientes:

***Tablas tiempo de ejecución:***

*NOTA: 0 corresponde a la carga de datos y los números son los requerimientos*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **RBT/CH** | **0** | **1** | **2** | **3** | **4** | **5** |
| **Small** | 143078 | 708.84 | 9331.8 | 19.1 | 18.26 | 4835.68 |
| **100000** | 205558 | 938.02 | 15481.4 | 23.66 | 21.58 | 6457.55 |
| **40000** | 88291 | 456.46 | 4531.59 | 12.9 | 14.72 | 2784.3 |

***Tabla 1. Tiempo de ejecución para arboles RBT y mapas de separate chaining***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BST/CH** | **0** | **1** | **2** | **3** | **4** | **5** |
| **Small** | 141175 | 711.85 | 9064.05 | 19.3 | 18.36 | 4822.13 |
| **100000** | 239739 | 946.11 | 15480 | 23.95 | 21.15 | 6485.21 |
| **40000** | 87255 | 447.34 | 4514.55 | 13.15 | 13.9 | 2839.54 |

***Tabla 2. Tiempo de ejecución para arboles BST y mapas de separate chaining***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BST/PR** | **0** | **1** | **2** | **3** | **4** | **5** |
| **Small** | 150812 | 719.38 | 8970.27 | 19.01 | 18.06 | 4900.87 |
| **100000** | 237080 | 956.32 | 15696.84 | 22.87 | 22.54 | 6508.43 |
| **40000** | 88037 | 458.53 | 4645.8 | 13.5 | 14.3 | 2803.9 |

***Tabla 3. Tiempo de ejecución para arboles BST y mapas con linear probing***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **RBT/PR** | **0** | **1** | **2** | **3** | **4** | **5** |
| **Small** | 143914 | 718.89 | 8884 | 18.9 | 17.21 | 4663.8 |
| **100000** | 204710 | 960.74 | 15536.66 | 23.02 | 21.78 | 6396.04 |
| **40000** | 86490 | 465.9 | 4509.76 | 12.82 | 13.9 | 2758.17 |

***Tabla 4. Tiempo de ejecución para arboles RBT y mapas con linear probing***

***Graficas de tiempos de ejecución:***

***Gráfica 1. Tiempo de ejecución de los requerimientos para los tipos de estructuras con el archivo small***

***Gráfica 1.1 Tiempo de ejecución de los requerimientos 3 y 4 para los tipos de estructuras con el archivo small***

***Gráfica 1.2 Tiempo de ejecución de la carga para los tipos de estructuras con el archivo small***

***Gráfica 2. Tiempo de ejecución de los requerimientos para los tipos de estructuras con el archivo de 100 mil reproducciones***

***Gráfica 2.1 Tiempo de ejecución de los requerimientos 3 y 4 para los tipos de estructuras con el archivo de 100 mil reproducciones***

***Gráfica 2.2 Tiempo de ejecución de la carga para los tipos de estructuras con el archivo de 100 mil reproducciones***

***Gráfica 3. Tiempo de ejecución de los requerimientos para los tipos de estructuras con el archivo de 40 mil reproducciones***

***Gráfica 3.1 Tiempo de ejecución de los requerimientos 3 y 4 para los tipos de estructuras con el archivo de 40 mil reproducciones***

***Gráfica 3.2 Tiempo de ejecución de la carga para los tipos de estructuras con el archivo de 40 mil reproducciones***

*Tablas consumo de memoria:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **RBT/CH** | **0** | **1** | **2** | **3** | **4** | **5** |
| **60000** | 4006365 | 429.23 | 40.46 | 8.8 | 994.65 | 112.77 |
| **100000** | 4169726 | 532.74 | 32.66 | 7.87 | 1192 | 172.59 |
| **40000** | 3670399 | 322.78 | 23.88 | 8.3 | 767.12 | 73.46 |

***Tabla 5. Consumo de memoria (kB) para árboles RBT y mapas de separate chaining***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BST/PR** | **0** | **1** | **2** | **3** | **4** | **5** |
| **60000** | 4006014 | 433.41 | 29.8 | 7.93 | 994.55 | 114.84 |
| **100000** | 4169426 | 534.31 | 34.73 | 7.35 | 1192 | 185.1 |
| **40000** | 3670049 | 318.01 | 19.75 | 4.78 | 767.12 | 74.5 |

***Tabla 6. Consumo de memoria para árboles BST y mapas con linear probing***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BST/CH** | **0** | **1** | **2** | **3** | **4** | **5** |
| **60000** | 4005955 | 433.93 | 27.22 | 32.49 | 996.24 | 97.38 |
| **100000** | 4169836 | 534.31 | 34.73 | 31.39 | 1192 | 174.52 |
| **40000** | 3670459 | 314.03 | 19.75 | 4.78 | 767.12 | 71.14 |

***Tabla 7. Consumo de memoria para árboles BST y mapas de separate chaining***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **RBT/PR** | **0** | **1** | **2** | **3** | **4** | **5** |
| **60000** | 4006424 | 429.26 | 28.25 | 8.45 | 994.15 | 100.63 |
| **100000** | 4169315 | 532.74 | 32.62 | 7.87 | 1192 | 170.94 |
| **40000** | 3669989 | 322.78 | 23.88 | 8.39 | 767.12 | 70.23 |

***Tabla 8. Consumo de memoria para árboles RBT y mapas con linear probing.***

*Gráficas de consumo de memoria:*

***Gráfica 4. Consumo de memoria de los requerimientos para los tipos de estructuras con el archivo small***

***Gráfica 5. Consumo de memoria de los requerimientos para los tipos de estructuras con el archivo de 100 mil reproducciones***

***Gráfica 6. Consumo de memoria de los requerimientos para los tipos de estructuras con el archivo de 40 mil reproducciones***

***Gráfica 7. Consumo de memoria una vez se carga la información de los archivos small***

***Gráfica 8. Consumo de memoria una vez se carga la información de los los archivos de 100 mil reproducciones***

***Gráfica 9. Consumo de memoria una vez se carga la información de los los archivos de 40 mil reproducciones***

En general, se puede ver que todos los requerimientos consumen una cantidad similar de memoria. Esto se debe a que en cada requerimiento una vez se obtiene la información deseada, el espacio que fue ocupado por todos los procesos para obtener la información es liberado o transformado a un tipo de dato de menor tamaño (None). Al momento de cargar, se espera que las tablas de Hash con “separate chaining” y los árboles BST ocupen menos memoria, lo cual se refleja en el consumo de memoria del archivo pequeño y en la gráfica 7, en la cual los árboles RBT y las tablas con “linear probing” presentaban mayor consumo de memoria.

*Nota: no consideramos que los valores de consumo de memoria de los otros archivos fueran a representar de manera fiel la teoría, ya que, estos fueron creados por nosotros y puede que las proporciones no sean iguales a través de todos los archivos de prueba).*

Como se puede ver, generalmente el tiempo de los requerimientos es menor cuando se consulta sobre un árbol RBT y dichos resultados de la consulta, suelen ser de acorde en la teoría, donde una búsqueda en un RBT asegura una complejidad de carácter logarítmico mientras que el árbol BST asegura un tiempo de búsqueda con orden de crecimiento lineal. En los requerimientos 1, 2, 3 y 5 vemos que, generalmente, al usar arboles RBT, los tiempos de respuesta son menores que al usar BST. No obstante, no es así en todas las pruebas, y atribuimos estas anomalías a dos factores distintos. Factores de la máquina o diferencias causadas por el tipo de tablas de Hash utilizadas. Teniendo en cuenta esto, se optó por utilizar los árboles RBT, ya que, la complejidad temporal de realizar una consulta sobre estos es ideal y aunque ocupan más memoria una vez cargada la información, consideramos que este costo vale la pena para tener mejores resultados en tiempo de procesamiento.

Para las tablas de Hash, aunque teóricamente el tiempo de consulta es bastante similar y se refleja claramente en el requerimiento 4. Nosotros observamos una mejora ligera en el tiempo de consulta sobre una tabla de hash que utilizaba el método de colisiones “linear probing”. Debido a que la mejora era ligera, solo usamos “linear probing” en las tablas de hash que requerían pocas llaves (valores menores a 50 llaves), sin embargo, cuando se requerían más de 1000 llaves, usamos “separate chaining” para reducir el espacio consumido en memoria después de realizar la carga de la información, sin tener mayores consecuencias en el tiempo de ejecución de cada requerimiento.

1. ***Conclusiones sobre los algoritmos usados para resolver los problemas.***

***2.1.***

Una buena comparación que se hizo (y así está implementado), fue la manera en la que, en una lista, quedaran elementos siempre sin repetir. Claramente, usar la función de comparación de una lista es la primera idea que debería llegar a la cabeza, pero, por alguna razón, las funciones de comparación que se hicieron no estaban funcionando de la manera esperada, entonces se tuvo que buscar otro camino.

Una solución encontrada fue el uso de la función “lt.isPresent()” de la librería de DISC. Esta sería una buena manera de solucionar el problema, si no fuera porque, cuando los elementos que compara son largos (30 o más caracteres), el tiempo de respuesta crece significativamente, aumentando también la complejidad del algoritmo. En realidad, cuando se usa con listas no tan grandes, no debería ser tan costoso el tiempo, pero una lista larga puede hacer crecer exponencialmente el tiempo de respuesta. A pesar de su asunto con el tiempo, la función “lt.isPresent()” no aumenta el uso de memoria, porque no necesita crear listas ni tablas externas.

La otra solución que se encontró fue recorrer la lista con los elementos repetidos e ir guardando cada elemento en una tabla de Hash externa, donde las llaves serían los elementos y los valores serían None. Esto solucionaría el problema del aumento de tiempo de respuesta, ya que, para tener la lista de elementos no repetidos, solo se necesita usar la función de DISC, “mp.keySet()”, la cual devuelve una lista con todas las llaves de una tabla. Como las llaves no están repetidas y acceder a una llave en una tabla de Hash es de orden constante, resulta beneficioso para los costos de procesamiento. Lo que sí aumentaría, sería el uso de memoria, por la razón que se creó una tabla de Hash aparte. Esto se puede solucionar reemplazando cada llave con None justo después de llamar a la función “mp.keySet()”, es decir, justo después de ya tener la lista con elementos no repetidos.