OBSERVACIONES DEL RETO 3

Samuel Alejandro Jiménez Ramírez – 202116652: Requerimiento 2
Marilyn Stephany Joven Fonseca – 202021346: Requerimiento 3

Ambiente de pruebas

	Máquina 1	Máquina 2
Procesadores	Intel® Core™ i5 - 10310U CPU @ 1.70 GHz 2.21GHz 64 bits	AMD Ryzen 5 4500U Radeon Graphics 2.38GHz
Memoria RAM (GB)	16 GB (15,6 utilizable)	16 GB
Sistema Operativo	Windows 10 Pro	Windows 10 Home

Tabla 1. Especificaciones de las máquinas para ejecutar las pruebas de rendimiento.

Resultado de pruebas

Para tomar el tiempo promedio por ejecución de cada requerimiento acá enlistado, usamos el archivo con terminación **large**. Tomamos 3 muestras de cada requerimiento y sacamos la media aritmética que es la que está escrita en la tabla.

Para la creación del catálogo, decidimos usar dos estructuras de datos al momento de cargar los datos en el programa, **tablas de hash** y **mapas ordenados**, el mecanismo de colisión para las tablas de hash que escogimos fue **linear probing** ya que es la mas eficiente que manejamos y los mecanismos para manejar los valores se acomodaban perfectamente a lo que buscábamos, por otro lado, los tipos de mapa que decidimos escoger fueron **RBT** ya que a pesar de que se demora mas la creación de estos este proceso solo se realiza al principio del programa por lo que es un costo que se puede aceptar.

```
catalog['info'] = om.newMap(omaptype="RBT", comparefunction=compareDates)
catalog['ciudad']=mp.newMap(numelements=804,maptype="LINEAR_PROBING",loadfactor=0.5) #ciudad-rbtavistamientos
catalog['topciudades'] = om.newMap(omaptype="RBT", comparefunction=compareDates) #rbt ordenado por cantidad de avistamientos.
catalog['hh:mm'] = om.newMap(omaptype="RBT", comparefunction=compareDates) #hora:min--arbolfecha-info
catalog['AA-MM'] = om.newMap(omaptype="RBT", comparefunction=compareDates) #A-M-D--arbolfecha-info
catalog['omxsegundos'] = om.newMap(omaptype="RBT", comparefunction=compareDates) #duration (seconds) - UFO
catalog["lat"] = om.newMap(omaptype="RBT", comparefunction=compareDates)
```

Para el mapa que se creo se tuvo en cuenta la cantidad de datos totales que tenemos de diferente valor en ciudades, ya que era la llave que esta iba a tener, se multiplicó por dos y se acercó al primo mas cercano.

Para las muestras no se va a medir la memoria utilizada puesto que esta es constante debido a la carga de datos que se realiza al comienzo del programa, y nunca se elevó a valores preocupantes que afectaran el buen funcionamiento de la máquina. Mismo caso que ocurrió con el procesador, nunca subió en ninguna de las dos máquinas a más del 25%.

Cabe destacar que el tiempo que se demora guardando datos del archivo csv con este tipo de estructuras de datos es de **19.09375s** en la **Máquina 1** y de **19.359375s** para la **Máquina 2**.

	Tiempo promedio Máquina 1 (s)	**Tiempo promedio Máquina 1 (s)**
Requerimiento 1	0.015625 s	0.0 s
Requerimiento 2	0.0 s	0.0 s
Requerimiento 3	0.0 s	0.0 s
Requerimiento 4	0.03125 s	0.015625 s
Requerimiento 5	0.015625 s	0.0 s
Requerimiento 6	0.015625 s	0.015625 s

Análisis complejidad temporal de cada requerimiento

Requerimiento 1:

Para el requerimiento 1 como hacemos una consulta sobre un map ya creado, el valor de la llave que buscamos es un rbt así que la complejidad temporal de este algoritmo sería **O(height)**, donde height es la altura del árbol que contiene la información de los avistamientos, dicha ciudad es escogida por el usuario por consola. Se obtiene esta complejidad ya que esa es la complejidad al buscar las mayores y menores llaves que es el proceso que realizamos mientras estamos buscando en el mapa. . Se hacen recorridos para crear listas o tomar valores, pero estos son despreciables o menores que esta complejidad.

Requerimiento 2:

Para el requerimiento 2 tenemos una complejidad de **O(height)** -**O(n)** donde n es la cantidad de datos a recorrer que se encuentran en el rango si este ocupa todos los datos, si no, la complejidad estaría dada por el height de el árbol resultante en este lapso dado por el usuario. Esta complejidad está dada por la función **om.keys y om.values** . Se hacen recorridos para crear listas o tomar valores, pero estos son despreciables o menores que esta complejidad.

Requerimiento 3:

Para el requerimiento 3 la complejidad temporal es **O(n)** donde n es la cantidad de datos que existen entre el rango inicial y final que dé el usuario, ya que se obtienen todas las llaves y se va haciendo un conteo del total de elementos para retornar el size. Se hacen otros recorridos pero para la muestra son despreciables al momento de dar la complejidad.

Requerimiento 4:

Para el requerimiento 4 la complejidad temporal es **O(n)** donde n es la cantidad de datos que se encuentran guardados en los arboles que están como valores en el rango dado por el usuario, ya que se hace una iteración a estos después de obtener los datos por un om.key y se recorren buscando los datos útiles para guardar en la tabla. Se hacen mas recorridos para crear listas mas pequeñas pero estas complejidades son despreciables.

Requerimiento 5:

Para el requerimiento 5 la complejidad asociada sería **O(n)** donde n es la cantidad de datos que se encuentran dentro del rango dado por el primer om.values, ya que después de tener las llaves se hace una iteración a esta lista resultante y dentro se hacen búsquedas a los mapas que tiene como valor. Por la manera en que guardamos los datos sabemos que no la complejidad de buscar en estos arboles puesto que los arboles son muy cortos ya que están distribuidos bien. Se hacen mas recorridos para crear listas de respuesta, pero estas son despreciables.

Requerimiento 6:

Para el requerimiento 6 la complejidad asociada sería O(n) al igual que en el requerimiento anterior, ya que se realiza una invocación a esta función para tomar los datos que se van a graficar, no sabemos la complejidad de graficar estos datos por eso dejamos esta complejidad.

Conclusiones

- 1. Se notó una mejoría implementando este tipo de estructura de datos, especialmente para obtener valores dentro de un rango establecido, a comparación de las tablas de hash o los mapas.
- 2. La carga de datos se torna un poco pesada debido a el tipo de estructura que se maneja, el cual es RBT, aunque esto nos garantiza una búsqueda mas constante, es importante saber que si se hubieran usado BST el tiempo de carga se reduciría notoriamente, pero no se aseguraría el mismo desempeño en el costo de tiempo que tiene cada uno de los requerimientos.
- 3. Es mas fácil acceder a un solo valor de la tabla con este tipo de estructura, también saber si existe o buscar un rango, ya que por la fórmula que tiene la altura del árbol, la que es N^2 donde N es la cantidad de datos, esto nos garantiza alturas muy pequeñas, lo que implica que las búsquedas también sean pequeñas en todos estos casos.