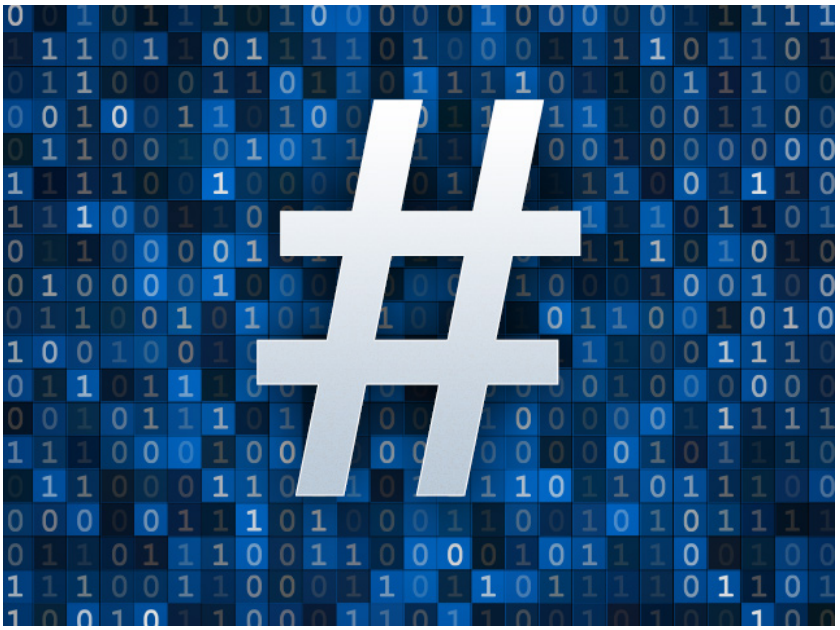
Act 5.2 - Actividad Integral sobre el uso de códigos hash (Evidencia Competencia)

Programación de estructuras de datos y algoritmos fundamentales

(Grupo 12)

Alumnos:
Marisol Rodríguez Mejía A01640086
Luis Enrique Lemus Martínez A01639698

Profesor:
Dr. Eduardo Arturo Rodríguez Tello



Para esta última actividad implementamos algo nuevo, que es el código hash, el cual es un identificador de 32 bits que se almacena en un Hash en la instancia de la clase. Toda clase debe proveer de un método hashCode() que permite recuperar el Hash Code asignado, por defecto, por la clase Object. El HashCode tiene una especial importancia para el rendimiento de las tablas hash y otras estructuras de datos que agrupan objetos en base al cálculo de los HashCode. (colaboradores de Wikipedia, 2020).

El hash es una técnica que se utiliza para identificar de forma única un objeto específico de un grupo de objetos similares.
En el hash, las claves grandes se convierten en claves pequeñas mediante el uso de funciones hash . Luego, los valores se almacenan en una estructura de datos llamada tabla hash . La idea del hash es distribuir las entradas (pares clave / valor) de manera uniforme en una matriz. A cada elemento se le asigna una clave (clave convertida).
Con una complejidad de O(1), la cual será muy eficiente al momento de compilarlo.
Una función hash es cualquier función que se puede utilizar para asignar un conjunto de datos de un tamaño arbitrario a un conjunto de datos de un tamaño fijo, que cae en la tabla hash. (Garg, 2016)

También el uso de las tablas hash fueron claves al momento de elaborar la actividad, una tabla hash es una estructura de datos que se utiliza para almacenar pares de claves / valores. Utiliza una función hash para calcular un índice en una matriz en la que se insertará o buscará un elemento. Al usar una buena función hash, el hash puede funcionar bien. Bajo supuestos razonables, el tiempo promedio requerido para buscar un elemento en una tabla hash es O (1) y máximo O(k), k siendo el número de colisiones
Asimismo, otras operaciones como eliminar e insertar elementos al inicio o al final de la tabla, son de poca complejidad temporal. Eliminar es O(k) e insertar O(1).

Para pasar de tener un archivo de texto a un grafo con todas las IPs y sus grados, leíamos el archivo y lo almacenamos en una lista de adyacencia. Dicha lista de adyacencia nos ayudó a manejar nuestras tablas hash durante el desarrollo de la actividad integral.

Para ordenar nuestra lista de adyacencia de menor a mayor dirección IP, utilizamos heapsort, el cual es un algoritmo de ordenamiento no recursivo, no estable, con complejidad computacional, el cual podríamos decir es uno de los más eficientes que hemos visto.

La verdad es que le sacamos mucho provecho a los grafos que utilizamos en la evidencia anterior, los cuales nos fueron de mucha utilidad al momento de implementar el código utilizando el hash code y las hash tables, las cuales nos apoyamos en actividades que hicimos previamente que nos sirvieron para esto.

Para la búsqueda de las IPs utilizamos el algoritmo de búsqueda binaria. Tiene una complejidad de O(log n), donde dicha complejidad se debe a que divide a la mitad el conjunto de entrada en cada iteración. Es más rápido que muchos otros tipos de algoritmos de búsqueda (Jindal, 2021).
El pseudocódigo del algoritmo es el siguiente:

busquedaBinaria(A, n, k)
Input: Un arreglo A ordenado de tamaño n, una llave de búsqueda k Output: El índice del primer elemento en A igual a k o -1 si no se encuentra $l \leftarrow 0$
$r \leftarrow n-1$
while $l \leq r$ do
$m \leftarrow l+(r-l)/2$; if $k == A[m]$ then
return m; else if $k < A[m]$ then
$r \leftarrow m-1$;
$l \leftarrow m-1$;
else
end end
return -1;

Cabe destacar el Max-Heap que usamos en esta actividad, el cual es una estructura de datos especializada basada en árboles que es esencialmente un árbol casi completo que satisface la propiedad del montón: en un montón máximo, para cualquier nodo C dado, si P es un nodo padre de C, entonces la clave (el valor) de P es mayor o igual que la clave de C. (colaboradores Wikipedia, 2021).

Al hacer el método que nos pidieron el cual nos pedían una IP solicitada por el usuario e imprimía una variedad de información pudimos entender la eficacia del código hash y sus beneficios.
También nos dimos cuenta que a lo largo de las evidencias implementamos métodos y estructuras de datos cada vez más eficientes, claro que hay algunos mejores que otros dependiendo del caso, pero la verdad es que el código hash es uno de los más eficientes y rápidos.

Por último, observamos el uso efectivo del código hash al resolver el problema y pudimos notar la eficiencia que tienen, por ello, fue enriquecedor implementarlos. Especialmente por la complejidad que tiene que para estas situaciones en que son una gran cantidad de datos como tenemos en el reto estas estructuras de datos nos son de mucha ayuda al momento de correrlo porque son muy eficientes.

La complejidad computacional de los métodos implementados se ve afectada por el número total de colisiones que se generan al agregar elementos a nuestro hash.
Utilizando la fórmula para el factor de carga:

$$\alpha = \frac{n}{m}$$

n siendo el tamaño de los datos en nuestro archivo, es decir, 13370 y m siendo el tamaño máximo de la tabla hash elegido, que en nuestro caso fue 25411.
Lo anterior lo sustituimos en la fórmula para calcular el número promedio de muestreos en las búsquedas no exitosas para obtener lo siguiente:

$$U = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \; U = \frac{1}{2}\left(1 + \frac{1}{\left(1-\frac{13370}{254111}\right)^2}\right) \qquad U = 2.72 \qquad \alpha = 52\%$$

Los anteriores valores de Uy de α explican la razón de por qué elegimos un tamaño de 25411. Podríamos haber elegido un número primo mayor, sin embargo, consideramos que debe de haber un balance entre un buen número de muestreos lineales y la memoria ocupada al crear la tabla hash. Mientras mayor sea el tamaño de la tabla, menos colisiones se generarán; no obstante, también es necesario tener en cuenta la memoria que estamos sacrificando para lograr lo anterior.
Debido a que tenemos 13,370 IPs, 12,041 elementos de nuestro hash quedan desocupados. Creemos que lo anterior no es un mal resultado, considerando que gracias a ese tamaño elegido, lograremos tener un menor número de colisiones al añadir elementos al hash. Nuestro promedio Utiene un valor de 2.72 y el factor de carga vale 0.52, los cuales son valores bastante buenos.

En una situación problema de esta naturaleza, el código hash nos puede ser de gran ayuda, complementándolo con los grafos. Lo anterior también debido a que son una estructura que nos permite marcar relaciones entre diferentes elementos, en este caso, las direcciones IP. Capturar las adyacencias a cada IP nos permitió manejar información como qué direcciones fueron las accesadas, y con la tabla hash nos fue posible guardar un resumen informativo sobre todas las IPs.

Referencias:

colaboradores de Wikipedia. (2020, 17 abril). HashCode() (Java). Recuperado 27 de noviembre de 2021, de https://es.wikipedia.org/wiki/HashCode()_(Java)

Garg, P. (2016, 26 abril). Basics of Hash Tables Tutorials & Notes | Data Structures. Recuperado 27 de noviembre de 2021, de

https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

Carlson, S. (s. f.). graph theory | Problems & Applications. Recuperado 23 de noviembre de 2021, de https://www.britannica.com/topic/graph-theory

colaboradores de Wikipedia. (2021, agosto 4). Grafo. Recuperado 22 de noviembre de 2021, de https://es.wikipedia.org/wiki/Grafo

GeeksforGeeks. (2021, 15 septiembre). HeapSort. Recuperado de https://www.geeksforgeeks.org/heap-sort/

Jindal, H. (2021, 11 marzo). Búsqueda binaria. Recuperado de https://www.delftstack.com/es/tutorial/algorithm/binary-search/#:%7E:text=Complejidad%20del%20algoritmo%20de%20b%C3%BAsqueda%20binaria,-Complejidad%20del%20tiempo&text=Cuando%20realizamos%20la%20b%C3%BAsqueda%20binaria,a%20la%20mitad%20cada%20vez.&text=Este%20resultado%20de%20esta%20recurrencia,orden%20de%20O(log%20n)%20.