CAMPUS QUERÉTARO

TC1031. PROGRAMACIÓN DE ESTRUCTURAS DE DATOS Y ALGORITMOS FUNDAMENTALES

(GRUPO 826)

"Evidencia 1: Actividades Integradoras - Resumen y Reflexión"

Evidencia presentada por la estudiante:

A01706095 - Naomi Estefanía Nieto Vega

Profesor:

Dr. Eduardo Arturo Rodríguez Tello

Fecha de entrega:

Lunes 7 de junio de 2021

Evidencia 1: Actividades Integradoras - Resumen y reflexión

A01706095 - Naomi Estefanía Nieto Vega

***Resumen*—A lo largo de este documento se expondrán las soluciones a las cinco actividades integradoras utilizando todos los temas vistos de las estructuras de datos y algoritmos fundamentales, desde los algoritmos de ordenamiento y búsqueda, las estructuras de datos lineales como los arrays, stacks, queues y listas ligadas, los árboles binarios de búsqueda o (BST), los grafos y las tablas hash, para concluir la solución que sería más eficiente en una situación problema de esta naturaleza. Asimismo se hablará acerca de las posibles mejoras para las soluciones presentadas.**

***Index Terms*—data structures, heap, array, bst, linked list, graph, hash tables, queue, stack, ip**

I. INTRODUCCIÓN

Hoy en día los algoritmos de ordenamiento y las estructuras de datos son muy importantes en el área computacional y en muchas más áreas en los que son utilizados ya que de acuerdo con la literatura un algoritmo es una serie o secuencia ordenada de pasos que dan solución a un determinado problema. No obstante, pueden existir distintos algoritmos que den solución a un mismo problema y es aquí donde entra la importancia de implementar el más eficiente en cuanto a tiempo y recursos ya que estos deben ser utilizados correctamente y de esta forma la computadora podrá darnos resultados muy buenos.

Asimismo, tener distintas opciones de algoritmos de ordenamiento nos da la posibilidad de mejorar la eficiencia de algún programa, utilizando el algoritmo que mejor solucione el problema y por consiguiente mejorar la eficiencia de nuestro programa al compilarlo, además de una adecuada utilización de los recursos ya que entre menos recursos utilice será mejor y más barata su implementación hablando económicamente en el impacto que tienen cuando desarrollamos software para una empresa.

Otro aspecto importante son los algoritmos de búsqueda ya que gracias a estos algoritmos es que podemos buscar datos sin tener que ir buscando uno por uno individualmente, tal vez esto parezca insignificante al utilizar una base de datos pequeña, pero cuando tenemos bases de datos con grandes volúmenes de datos esto se volvería algo tedioso, es por ello que su importancia radica en que nos facilitan la búsqueda de datos de una forma eficiente. Además de estos algoritmos existen otras estructuras de datos que son un poco más complejas pero se vuelven más eficientes al manejar grandes cantidades de datos o información y querer realizar búsquedas, como es el caso de las tablas hash.

Así, al conocer la importancia y eficiencia del uso de las estructuras de datos y sus implementaciones podemos desarrollar opciones viables en cuanto a tiempo y eficacia para búsquedas, inserción, borrado y demás funciones en bases de datos con archivos muy grandes, como la de la situación problema, que consiste básicamente en una bitácora con registros de accesos de varias direcciones IP.

II. DESARROLLO

II-A. Actividad Integradora 1. Conceptos básicos y algoritmos fundamentales

Para los algoritmos de búsqueda tenemos dos opciones, la búsqueda secuencial que en su mejor caso cuenta con una complejidad de C(n) = 1 si el elemento buscado es el primero de la lista, y en su peor caso cuenta con una complejidad de C(n) = n si el elemento buscado no existe o se encuentra en la última posición. La segunda opción es la búsqueda binaria, cabe aclarar que ambos métodos realizan lo mismo pero con un algoritmo distinto, en este caso utilicé la búsqueda binaria en un arreglo ya ordenado que cuenta con una complejidad de O(log n) para su peor caso y un O(1) en su mejor caso. A continuación podemos observar una tabla con algunos de los algoritmos de ordenamiento:

Algoritmo	Mejor	Promedio	Peor	Estable	Espacial
SwapSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Figura 1. Complejidades de los algoritmos de ordenamiento vistos a lo largo de estas cinco semanas.

Como podemos observar en la figura 1, tenemos que los algoritmos de SwapSort y SelectionSort son muy similares en cuanto a su complejidad espacial y temporal, a excepción de la estabilidad, lo que significa que por ejemplo el algoritmo de SelectionSort si llegaran a existir dos objetos con la misma clave después del ordenamiento no se respetaría el mismo orden, a diferencia del SwapSort. Dado que para ambos su mejor complejidad es de O(n^2) no los consideraría una opción viable para los requerimientos de esta actividad.

Posteriormente tenemos a BubbleSort e InsertionSort que son muy similares en cuanto a complejidad, pero no consideré que fueran lo suficientemente aptos para este tipo de búsqueda tratándose de archivos muy grandes. Después tenemos a MergeSort y QuickSort, que a simple vista parecería mejor MergeSort que QuickSort debido a su complejidad. Sin embargo, para el desarrollo de esta primera actividad

integradora utilicé el algoritmo de ordenamiento "QuickSort" y a continuación explicaré los motivos por los cuales consideré a este algoritmo como la mejor opción de los algoritmos de ordenamiento vistos:

- Es un algoritmo basado en la técnica de divide y vencerás por lo que funciona eligiendo un pivote para particionar el arreglo con base en esto y así ir ordenando los elementos.
- No necesita espacio auxiliar de almacenamiento ya que es un algoritmo de ordenamiento in situ, esto quiere decir que no necesita espacio de memoria auxiliar para hacer el ordenamiento, a diferencia de "MergeSort" por ejemplo, en el que tenemos que crear dos arreglos temporales para fusionar las mitades.
- Es un buen algoritmo de ordenamiento cuando estamos tratando con arreglos o vectores aleatorios, ya que a pesar de tener una complejidad de O(n^2) en su peor caso, existe una probabilidad alta de que con el pivote correcto se vuelva muy eficiente, siendo su mejor caso un O(n log n).
- Es un algoritmo que maneja estabilidad, esto quiere decir que si tenemos dos objetos con la misma clave aparecerán en el mismo orden en la salida arrojada.

De acuerdo con los motivos señalados anteriormente es que decidí implementar este algoritmo, además de que al cronometrar el tiempo que dicho algoritmo tarda en realizar el ordenamiento de los datos depende mucho del pivote como mencioné anteriormente, por lo que al tener un buen pivote el tiempo de ejecución se reduce significativamente. Su rango de tiempo al compilarlo osciló entre los 7 a 24 milisegundos en su peor caso (basándome en las pruebas al compilar en consola). Este tiempo es muy variable y no es definitivo. Como podemos observar en la figura 3, este algoritmo tiene una complejidad promedio de O(n log n), esto quiere decir que los datos no siguen algún patrón que aporte ventajas o desventajas, por lo que se considera la situación típica de ejecución.

Ahora procederemos a ver el análisis de la complejidad del algoritmo QuickSort. La complejidad de este algoritmo de ordenamiento podemos expresarla de la siguiente manera:

$$T(n) = T(k) + T(n-k-1) + O(n) \tag{1}$$

Donde, k es el número de elementos menores al pivote

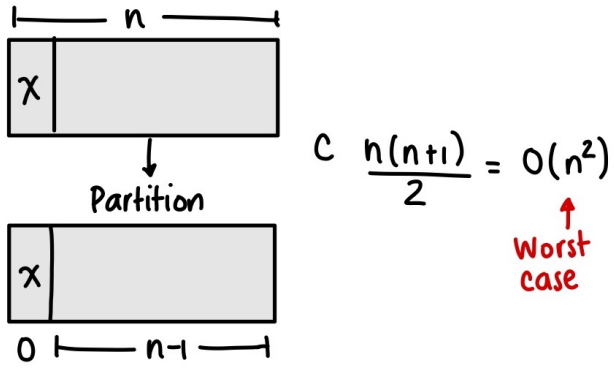

Figura 2. Ilustración del peor caso del algoritmo Quicksort.

Entonces analizando el peor caso de este algoritmo, sabemos que sería cuando tenemos un arreglo que ya está ordenado, ya que nuestra x estaría hasta la izquierda como se puede observar en la figura 4. Y por consiguiente, al dividir las ramas tendríamos constantes que van de la siguiente forma,

$$C\,[n + (n-1) + (n-2) + ..., 1]$$

por lo que después pasaría a ser una suma de Gauss, definida por la siguiente ecuación,

$$\left(\frac{n(n+1)}{2}\right)$$

y la ecuación resultante de esto nos da la complejidad obtenida en la ecuación 2,

$$T(n) = O(n^2) \tag{2}$$

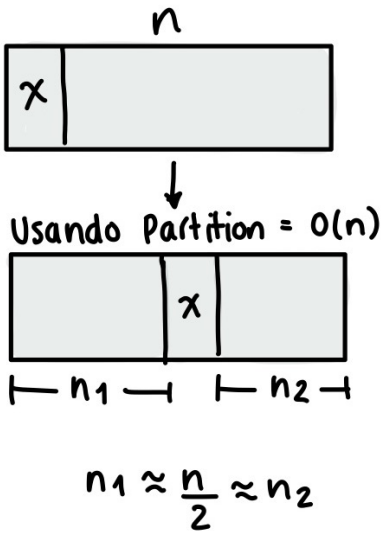

Figura 3. Ilustración del mejor caso del algoritmo Quicksort.

En cambio para su mejor caso tenemos un arreglo o vector de tamaño n, nuestro pivote en este caso es la x como se muestra en la figura 5, y si al momento de hacer la partición

(con complejidad de O(n)), se divide en dos partes, nuestras ecuaciones quedarían de la siguiente manera,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + Cn$$

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn$$

$$T(n) = O(\log n) \tag{3}$$

Por lo tanto, su complejidad espacial en el mejor caso sería la obtenida en la ecuación 3.

II-B. Actividad Integradora 2: Estructuras de Datos Lineales

Para los algoritmos de búsqueda tenemos dos opciones, la búsqueda secuencial que en su mejor caso cuenta con una complejidad de C(n) = 1 si el elemento buscado es el primero de la lista, y en su peor caso cuenta con una complejidad de C(n) = n si el elemento buscado no existe o se encuentra en la última posición. La segunda opción es la búsqueda binaria, cabe aclarar que ambos métodos realizan lo mismo pero con un algoritmo distinto, en este caso utilicé la búsqueda binaria en un arreglo ya ordenado que cuenta con una complejidad de O(log n) para su peor caso y un O(1) en su mejor caso.

Sin embargo, para el desarrollo de esta actividad integradora utilicé el algoritmo de ordenamiento «QuickSort» y a continuación explicaré los motivos por los cuales consideré a este algoritmo como la mejor opción de los algoritmos de ordenamiento vistos:

- Es un algoritmo basado en la técnica de divide y vencerás por lo que funciona eligiendo un pivote para particionar el arreglo con base en esto y así ir ordenando los elementos.
- No necesita espacio auxiliar de almacenamiento ya que es un algoritmo de ordenamiento in situ, esto quiere decir que no necesita espacio de memoria auxiliar para hacer el ordenamiento, a diferencia de «MergeSort» por ejemplo, en el que tenemos que crear dos arreglos temporales para fusionar las mitades.
- Es un algoritmo que es estable, esto quiere decir que si tenemos dos objetos con la misma clave aparecerán en el mismo orden en la salida arrojada.
- Aunque sabemos que en el caso de las listas doblemente ligadas un mejor algoritmo para esto sería el «MergeSort», podemos adaptar el algoritmo de «QuickSort» para que funcione de forma iterativa, que si bien sabemos esto no altera su complejidad ya que la esencia del algoritmo es la misma, podemos notar que mejora en cuestión de eficacia de uso de memoria de acuerdo con la complejidad espacial, ya que en este caso nosotros manipulamos el stack.

Las estructuras de datos son formas de organización y representación de datos o información, en el caso de la programación, son formas de organización que nos permiten utilizar estos datos de una forma efectiva. Existen dos tipos de estructuras de datos:

- Estructuras de datos lineales
- Estructuras de datos no lineales

Su principal diferencia es que la estructura de datos lineal tiene a sus elementos de datos dispuestos de manera secuencial y cada uno está conectado con su elemento anterior y siguiente. Por otra parte, las estructuras de datos no lineales no tienen una secuencia establecida para conectar a sus elementos, por lo que un elemento puede tomar varias rutas para conectarse con otros elementos. Este tipo de estructuras son un poco más complejas de implementar pero son más eficientes en cuanto a uso de memoria en la computadora. [2] En la figura 4 podemos observar de manera gráfica la clasificación de los tipos de estructuras de datos.

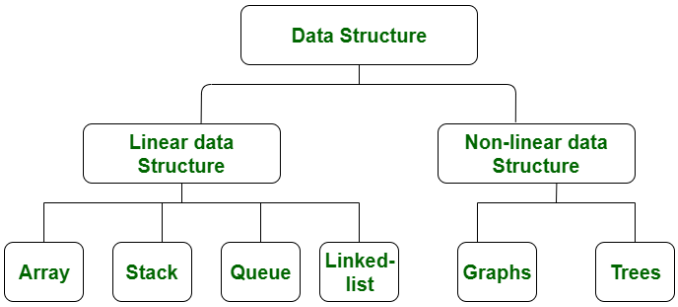

Figura 4. Clasificación de las estructuras de datos. Imagen obtenida de: geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/

Ahora bien, para la solución de esta segunda actividad integradora fue muy importante el uso de estructuras de datos lineales que como sabemos son estructuras que tienen elementos conectados de manera secuencial de forma que el elemento está conectado a su elemento anterior y al próximo, además son estructuras relativamente sencillas de implementar pues la naturaleza de la memoria de las computadoras también es secuencial. [2] Algunas estructuras de datos de esta naturaleza son las listas ligadas (Linked List), colas (Queue), pilas (Stack) y arreglos (Array). La importancia de las estructuras de datos lineales radica en que gracias a ellas podemos hacer más eficiente la organización de datos y la forma en que posteriormente utilizamos o analizamos estos datos. Además nos permiten integrarlas en conjunto con algoritmos de ordenamiento de forma que podemos mejorarlo en términos de eficiencia.

Para este caso en particular utilicé nuevamente el algoritmo de Quicksort en su versión iterativa, por lo que su complejidad algorítmica y espacial permanece siendo la misma (ver figura 5), ya que la esencia del algoritmo es la misma y trabaja de la misma forma que la versión recursiva, la única diferencia es que en la implementación iterativa manejamos explícitamente el stack a diferencia de la implementación recursiva en la que dependemos del stack que viene implícito.

Mejor	Promedio	Peor	Estable	Complejidad espacial
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Figura 5. Complejidad algorítmica del algorítmo «QuickSort».

Otra modificación que fue realizada al algoritmo es en los parámetros de entrada, ya que el anterior recibe un arreglo y

ahora recibe una lista doblemente ligada (Doubly Linked List).

Las listas ligadas son un tipo de estructura de datos lineal en la que los elementos no son ordenados o almacenados en localidades temporales de memoria contiguas, sino que están ligados mediante el uso de apuntadores (Ver figura 6) [4]. Sabemos que existen tres principales tipos de listas ligadas (Linked List):

- Lista ligada simple
- Lista doblemente ligada
- Listas circulares

La principal diferencia entre la lista ligada simple y la doblemente ligada es que en la simple únicamente tenemos un apuntador o referencia al siguiente nodo. En cambio, en la doblemente ligada además del apuntador hacia el siguiente nodo tenemos otro extra que apunta hacia el nodo anterior y esto nos da la posibilidad de volver más eficientes nuestras funciones al momento de implementarlas.

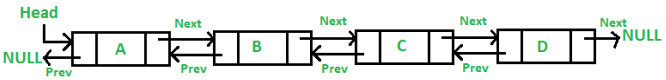

Figura 6. Funcionamiento de las listas doblemente ligadas. Imagen obtenida de: geeksforgeeks.org/doubly-linked-list/

Por lo tanto, para la solución de esta actividad integradora es preferible la implementación de las listas doblemente ligadas (Doubly Linked List) porque gracias a que tenemos un apuntador que apunta hacia el final de la lista o el «tail» además del «head», es posible mejorar la complejidad algorítmica de nuestra implementación y reducir de un O(n) a un O(1) que es mucho mejor.

Algunas de las funciones básicas de las listas doblemente ligadas son las siguientes:

- Insertion
- Deletion
- Search

Dichas funciones son muy importantes porque con ellas principalmente es que podemos comenzar a manejar datos en nuestra Doubly Linked List, para insertar, eliminar o buscar datos. La complejidad de estas funciones en parte depende de su implementación, ya que dependiendo de si tenemos un tail su complejidad mejora. Por ejemplo, para el insertion si es al principio de la lista su complejidad es de O(1), en cambio si es para añadir al final su complejidad con un apuntador a tail es de O(1) ya que reduce significativamente la cantidad de procesos que hay que realizar dentro de la función y en su peor caso es de O(n).

Para la función de borrado, en su mejor caso tiene una complejidad de O(1) y en su peor caso una complejidad de O(n) porque debe iterar en toda la lista y comparar por cada nodo la posición o el valor del dato y detenerse hasta que la condición de que el valor sea igual al valor del nodo en ese momento para proceder a eliminarlo. Y por último la función de búsqueda tiene una complejidad de O(n) en ambos casos porque tiene que iterar en la lista y por cada nodo que exista debe comparar si el valor del dato es igual al solicitado y hasta que se cumpla la condición es que termina esta función, entonces esto nos genera una complejidad espacial de O(n). En la figura siguiente se puede observar un resumen de las complejidades de cada función.

Función	Promedio	Peor
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(n)$
Search	$O(n)$	$O(n)$

Figura 7. Complejidades de las principales funciones de las listas ligadas.

Para concluir, el desarrollo de estas funciones y su implementación son muy importantes para el desarrollo de la solución ya que una implementación no tan acertada se traduce en un uso ineficiente de los recursos como lo es la memoria. Es por ello que se debe cuidar la manera de implementar cada una y hacerlas lo más eficientes posibles de acuerdo con los requerimientos solicitados.

II-C. Actividad Integradora 3: Estructura de Datos Jerárquica

A continuación, en la siguiente figura podemos observar un gráfico con las complejidades (temporal y espacial) de las distintas estructuras de datos que existen.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figura 8. Complejidades de las estructuras de datos más comúnes. Obtenido de: https://www.bigocheatsheet.com/

Los Árboles Binarios de Búsqueda (o Binary Search Tree (BST), por sus siglas en inglés) son una estructura de datos jerárquica basada en nodos donde cada nodo tiene una clave (key) y un valor asociado. [3] Permite operaciones como búsqueda, añadir, y eliminar elementos de forma muy rápida y eficaz. Los nodos se organizan de acuerdo con las siguientes características:

- El subárbol izquierdo de un nodo contiene sólo nodos con claves menores que la clave del nodo.
- El subárbol derecho de un nodo contiene sólo nodos con claves mayores que la clave del nodo.
- Los subárboles izquierdo y derecho deben ser un BST.

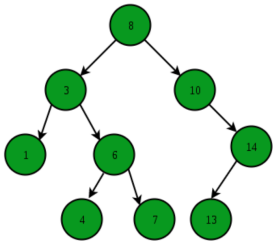

Figura 9. Representación de un árbol binario de búsqueda. Obtenido de: https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Esta estructura de datos es capaz de representar a los datos en una estructura jerárquica y es comúnmente utilizada en muchas aplicaciones de búsqueda en las que los datos entran y salen constantemente del mapa. Los BST tienen una complejidad temporal promedio de O(log n) y sus principales funciones como búsqueda, inserción y borrado tienen en promedio una complejidad de O(log n) y en su peor caso tienen una complejidad de O(n). En la figura 10, podemos observar un resumen de las complejidades de las operaciones mencionadas anteriormente.

Operación	Promedio	Peor
Buscar	O(log n)	O(n)
Insertar	O(log n)	O(n)
Borrar	O(log n)	O(n)

Figura 10. Complejidades temporales de las principales operaciones de los BST.

Los BST son importantes en la situación problema porque gracias a ellos podemos buscar de una forma muy eficiente en la bitácora con las direcciones IP por sus accesos dependiendo y dependiendo de cuántas veces aparece la misma IP será la cantidad de accesos. En este caso para determinar si una red está infectada por la cantidad de accesos en cada IP tendríamos que analizar la respuesta del servidor, tomando como "infectada" a aquellas direcciones IP que han regresado algún error al momento del acceso, de esta forma gracias a las operaciones que nos es posible realizar con los BST podemos buscar eficientemente las IP que posiblemente han sido infectadas debido a la cantidad de accesos concurrentes y que además tienen algún mensaje de error como respuesta. Es por ellos que los BST son bastante eficaces y útiles en situaciones de esta naturaleza.

II-D. Actividad Integradora 4: Grafos

Los grafos son una estructura de datos conformada por un conjunto finito de elementos o nodos (vértices) y un conjunto de aristas que los conectan. Se considera como una arista a un par (a,b) que nos dice que el vértice a está conectado con el vértice b. [3] Existen dos tipos de grafos:

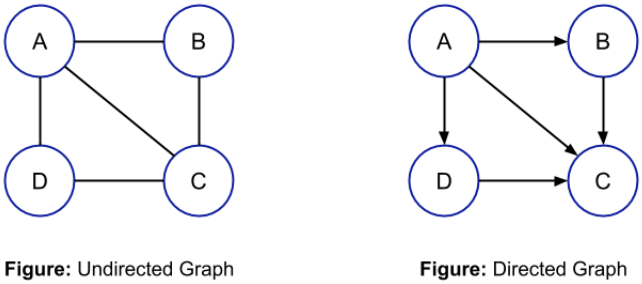

Figure: Undirected Graph **Figure:** Directed Graph

Figura 11. Tipos de grafos. Obtenido de: https://afteracademy.com/blog/introduction-to-graph-in-programming

- Grafo no dirigido

En este tipo de grafo los nodos se conectan por bordes bidireccionales, esto quiere decir que se puede ir de un nodo a otro y viceversa. Por ejemplo, podemos atravesar del nodo A al nodo B y del nodo B al nodo A.

- Grafo dirigido

Por otra parte, en los grafos dirigidos los nodos se conectan por bordes que son dirigidos, es decir, van en una sola dirección. Por ejemplo, si hay una conexión entre el nodo A y B pero la flecha apunta hacia B, significa que ese nodo sólo se puede atravesar de A hacia B y no en la dirección opuesta.

Asimismo los dos tipos de representaciones de grafos más comunes son:

- Lista de adyacencias

La lista de adyacencias se crea utilizando una matriz de listas en la que el tamaño de la matriz es igual al número de nodos o vértices, además un único índice representa la lista de nodos adyacentes al i-ésimo nodo. Se pueden usar para representar grafos ponderados y los pesos de sus aristas como listas de pares (a, b).

Las ventajas que tienen las listas de adyacencias es que ahorran mucho espacio ya que tienen complejidades de O(|V| + |E|) y en el peor caso puede existir un número C(V, 2) de aristas que consuma un espacio de O(V^2). Además agregar vértices es más sencillo. Por otra parte las desventajas podrían ser que las consultas como checar si hay una conexión desde un vértice A hasta el vértice B no son muy eficientes y manejan complejidades de O(V).

- Matriz de adyacencias

La matriz de adyacencia es una matriz 2D de tamaño VxV en donde V es el número de vértices del grafo, entre sus características sabemos que para el caso de los grafos no dirigidos esta matriz siempre es simétrica y también se utiliza para representar grafos ponderados. [4]

Entre las ventajas que tiene la matriz de adyacencias es que su representación es más fácil de implementar, y que sus funciones para eliminar una conexión o realizar una consulta del vértice A hasta el B se pueden realizar en O(1), por otro lado su desventaja podría ser que aún así el grafo consume más espacio (aunque tenga menos aristas), y que agregar un vértice tiene una complejidad de O(V^2). [4] Sus principales funciones tienen las siguientes complejidades como se puede observar en la Figura 2. Además de todo esto los grafos también pueden tener peso o no tenerlo.

Graph Data Structure Operations

Data Structure	Time Complexity					
	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Figura 12. Complejidades de las principales funciones de grafos. Obtenida de: https://stackoverflow.com

Cabe señalar que esta estructura de datos es más compleja de utilizar pero se utiliza mucho para resolver problemas de la vida real, por ejemplo: en las conexiones entre perfiles en las redes sociales, las redes telefónicas, los mapas, entre otros.

La importancia de los grafos radica en que son formas de visualizar relaciones entre datos y su principal propósito es representar información que es muy grande en volumen de datos o muy compleja para describir en texto o menos espacio. Esto nos facilita a nosotros como personas la visualización de esta información pero además podemos añadir funciones que con base en ciertas variables nos den información valiosa, como lo es el caso de Google Maps, que calculan la mejor ruta con base en la distancia (kilómetros) entre dos vértices. Similar es el caso de Facebook, en el que los usuarios son vértices nodos y las sugerencias de amigos se basan en las conexiones que hay entre nodos. Esto permite hacer más eficientes los procesos de relación de información y sacar el máximo provecho de esta.

II-E. Actividad Integradora 5: Uso de Hash Tables

Las tablas hash o mejor conocidas como hash tables en inglés, son una estructura de datos que almacena datos de forma asociativa, esto quiere decir que se almacenan en un tipo o formato de matriz en la que cada valor de datos cuenta con un key o índice único, de esta forma el acceso a los datos se vuelve muy eficiente y rápido si conocemos la llave de los datos deseados.

Por otra parte, también existe la función de hashing, que en la mayoría de los casos funciona muy bien a comparación de otras estructuras de datos como Arrays, Linked List o BST, ya que con el hashing tenemos una complejidad de O(1) en tiempo de búsqueda en promedio y un O(n) en el peor de los casos.

La función hash es una función que asigna un número grande o una cadena a un entero pequeño que se puede utilizar como índice en la tabla hash. Una función eficiente de hash tendría las siguientes propiedades:

- Eficientemente computable
- Distribuye uniformemente las llaves (cada posición de la tabla es igualmente probable para cada llave)

Por otra parte, dada una función hash nos da un número pequeño para una llave grande, existe la posibilidad de que dos llaves den el mismo valor, cuando una llave recién asignada se asigna a un elemento ya ocupado en la tabla hash se le conoce como colisión, y para ello existen técnicas de manejo de colisiones:

- Encadenamiento (Chaining)

Esta técnica consiste en hacer que cada celda de la tabla hash apunte a una lista vinculada de registros que tengan el mismo valor de función hash. Esta técnica es simple pero requiere memoria adicional fuera de la tabla.

- Direccionamiento abierto (Open Addressing)

Por otra parte, en el direccionamiento abierto todos los elementos se almacenan en la propia tabla hash, de esta forma cada entrada de la tabla tiene un registro, y al buscar un elemento se va examinando uno por uno cada espacio de la tabla hasta encontrar el elemento deseado o no encontrarlo en caso de que no esté en la tabla. [3]

Entonces, hablando en términos de la eficiencia, las operaciones de inserción y búsqueda son muy rápidas independientemente del tamaño de los datos, ya que este tipo de estructura lo que hace es usar una hash table como matriz de almacenamiento y con la técnica de hashing genera los keys o índices únicos en el cual se insertan los datos. [2] En la siguiente figura (fig. 13) se puede observar una tabla con las complejidades de las principales funciones mencionadas anteriormente.

Hash table	Average	Worst
Search	O(1)	O(n)
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)

Figura 13. Complejidad de las principales funciones de las hash tables.

La importancia de las hash tables en una situación problema de esta naturaleza es que este tipo de estructura de datos es muy eficiente cuando tenemos que buscar algún elemento específico en una bitácora o set de datos muy grande, ya que tienen una complejidad de O(1). [4] Por lo que para el caso de la búsqueda de las IPs adyacentes a una IP dada por el usuario se convierte en una estructura ideal y bastante rápida para encontrar la información. Además son muy utilizadas en particular para matrices o arrays asociativos, indexación de bases de datos, cachés y conjuntos.

III. CONCLUSIÓN

En conclusión, después del análisis de todas las implementaciones con los distintos algoritmos de algunas estructuras de datos, las soluciones que considero más eficientes son la de grafos y hash tables; los grafos principalmente porque son una forma eficiente de organizar y relacionar información, además entre sus ventajas está que pueden tener peso o no, y con esto podemos añadir variables a considerar, por ejemplo en el caso de los mapas en la cual no sólo se considera la distancia sino también los delays y tráfico en la ruta, entre otras factores. Gracias a esto es posible que el algoritmo detecte rutas o vías más rápidas. Para el caso de la situación problema en cuestión fue de mucha utilidad al momento de querer conocer los nodos o vértices con los que conectaba cada IP.

Por otro lado, las hash tables son un método más viable para la recolección de datos que cualquier otra estructura, esto debido a que la búsqueda en un hash es mucho más rápida que en un arreglo o una lista, sin embargo, no es

igualmente eficiente para realizar ordenamiento de datos, pero sí son buenas para realizar búsquedas en archivos con grandes volúmenes de datos ya que manejan una complejidad constante O(1) y por lo tanto encuentran la información bastante rápido. Por ejemplo, en el caso de la situación planteada es muy poco tiempo el que toma encontrar todas las direcciones IP que son adyacentes a una IP dada.

En cambio, las soluciones que aún podría mejorar para que fueran más eficientes son la primera en la parte de crear las llaves para cada IP, ya que en esta en particular lo que hice fue concatenar la fecha y la hora para que al momento de realizar la búsqueda ya estuviera ordenado y esto funcionó, sin embargo ahora que conozco más sobre otras estructuras de datos más complejas como lo son los grafos o tablas hash, me doy cuenta que la implementación de la primera solución se podría mejorar y se evitarían problemas de escalabilidad si en un futuro se deseara considerar el año, además de la fecha y hora. La forma en la que implementaría esta mejora sería con ayuda de los hash hables, o bien, utilizando la librería del formato para fechas en C++, ya que con esta librería se podría hacer un manejo más eficiente de las fechas y sin tanta preocupación al momento de generar el ID.

REFERENCIAS

[1] Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. (2020). Big O Algorithm Complexities. https://www.bigocheatsheet.com/

[2] GeeksforGeeks. (2019, 30 septiembre). Complexity Analysis of Binary Search. https://www.geeksforgeeks.org/complexity-analysis-of- binary-search/

[3] GeeksforGeeks. (2021, 27 febrero). QuickSort. https://www.geeksforgeeks.org/quick-sort/

[4] GeeksforGeeks. (2019a, septiembre 26). Stability in sorting algorithms. https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

[5] Time complexity:Best and Worst cases|Quick Sort|Appliedcourse.(2019, 18 mayo).[Vídeo].YouTube. https://bit.ly/3c9VzPr

[6] GeeksforGeeks. (2021a, enero 4). Why quicksort is better than mergesort ? https://www.geeksforgeeks.org/quicksort-better-mergesort/

[7] Difference between Linear and Non-linear Data Structures. (2021). Tuto- rialspoint. https://www.tutorialspoint.com/difference-between-linear-and- non-linear-data-structures

[8] GeeksforGeeks. (2020a). Linked List Data Structure. https://www.geeksforgeeks.org/data-structures/linked-list/

[9] GeeksforGeeks.(2021,9marzo).DoublyLinkedList|Set1(Introduction and Insertion). https://www.geeksforgeeks.org/doubly-linked-list/

[10] LinearLinkedList:TimeandSpaceComplexityofInsertionandDeletion. (2020, 9 mayo). [Vídeo]. YouTube. https://bit.ly/3wPn2hx

[11] GeeksforGeeks. (2020). Binary Search Tree. https://www.geeksforgeeks.org/binary-search-tree-data-structure/

[12] Edpresso Team. (2021, 23 abril). What is a graph (data structure)? Educative: Interactive Courses for Software Developers. https://www.educative.io/edpresso/what-is-a-graph-data-structure

[13] GeeksforGeeks. (2021, 8 marzo). Graph and its representations. https://www.geeksforgeeks.org/graph-and-its-representations/

[14] Introduction to Graph in Programming. (2021). AfterAcademy. https://afteracademy.com/blog/introduction-to-graph-in-programming

[15] Uzgalis, R. (1995) Advantages of Hash Search. Random Numbers, Encryption, and Hashing. Recuperado de: http://www.serve.net/buz/Notes.1st.year/HTML/C6/rand.016.html el día 03 de junio de 2021.