Act 6.2 - Reflexión Final de Actividades Integradoras de la Unidad de Formación TC1031 (Evidencia Competencia)

Programación de estructuras de datos y algoritmos fundamentales (Gpo 4)

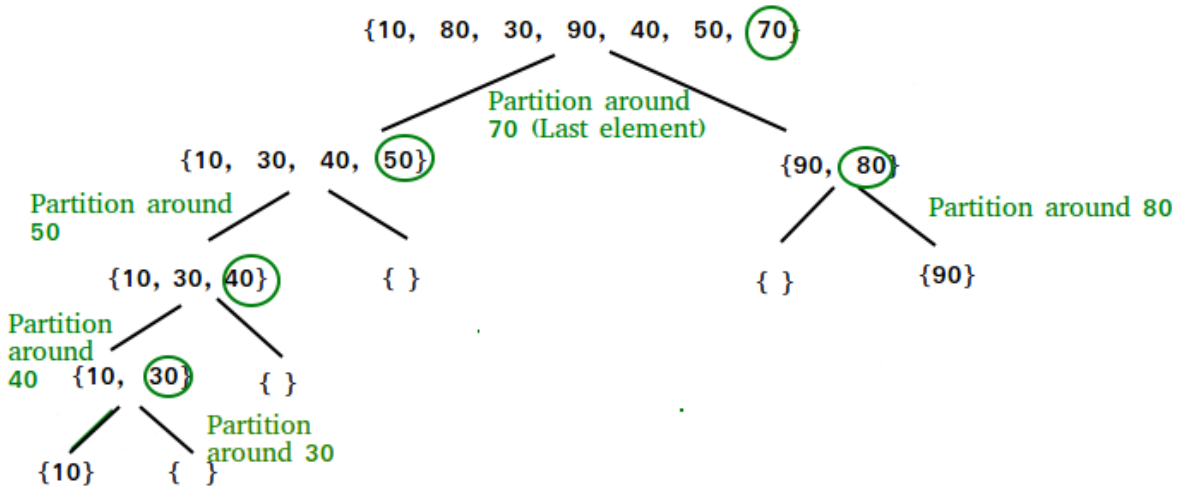
Thomas Freund Paternostro

A00831997

Fecha de entrega:

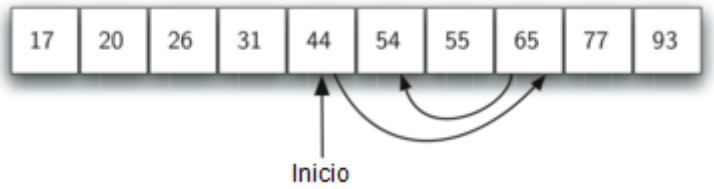
24/11/2021

Reflexión 1.3-



Explicación Quicksort: Este es un algoritmo que por medio de recursividad hace un llamado a otra función que hace particiones donde el objetivo es dividir y vencer donde se puede resolver un problema en nlogn hasta n^2 como fue demostrado en la parte superior. En la imagen se puede evidenciar cómo dividir las particiones y por medio de esto identifica el mayor y los agrupa de nuevo en la función de particiones. (GeeksforGeeks)

Explicación Partición : Esta función toma el valor inicial que sería cero y el final que es n-1. Donde se intercambian los valores si el valor almacenado es menor y este se repetirá n veces dado que tiene un loop for que va de 0 a n-1 que seria n veces dado que el último término automáticamente llegaría a ser el mayor.



Explicación Búsqueda Binaria: La búsqueda binaria utiliza el loop while como un recurso donde el valor mínimo y máximo se está aproximando a uno exacto que es el que se está buscando.

Reflexión:

Se consideró trabajar con mergesort, pero no veiamos ventajas a su uso sobre quicksort y este hacia el trabajo necesario de una manera más eficiente. La búsqueda binaria fue uno de los aspectos más difíciles de la actividad dado que se tuvo que pensar en un sistema donde se

podía buscar hasta el segundo (aunque no fue requerido) para tener un programa más completo.

A lo largo de esta actividad se pudo trabajar con los diferentes algoritmos que fueron estudiados esta semana. Entender más a fondo cómo utilizarlos en contextos prácticos y tener que modificar los para poder obtener resultados precisos y llegar a una solución esperada. Se pudo entender a mayor profundidad su importancia la importancia de la eficiencia de cada uno y entender cuando aplica cada uno y por qué debería ser descartada la opción de algunos sobre otro dependiendo del caso. Se trabajó con dos algoritmos que son quicksort y búsqueda binaria.

Reflexión 2.3-

Incluir de manera específica si el uso de un lista doblemente ligada para esta situación problema es la más adecuada o no así como sus ventajas y desventajas

La lista doblemente ligada tiene como ventaja poder retroceder y moverse a lo largo de esta, tanto de principio a fin como de fin a principio, además de colocar y crear memoria acorde a la necesidad que se requiera, sin acaparar más de la necesaria. Se trabajó para hacer el arreglo utilizando un algoritmo quicksort, el cual en el caso promedio es O(n log n) y el peor caso es O(n^2). No consideramos que en el peor caso de la función Partición que tiene complejidad O(n) afecte la eficiencia o procesamiento de los archivos de texto. En donde claramente estamos trabajando con un algoritmo deficiente, sin embargo encontramos que para la carga de 16807 IPs, la velocidad de ordenamiento supera lo esperado.

La gran desventaja de las listas ligadas es el tiempo que toma acceder a uno de sus elementos por índice. Para, por ejemplo, acceder al quinto elemento de la lista, tendríamos que iterar 5 veces, accediendo a los 4 nodos anteriores al quinto.

Aunque, aprovechando que se trata de una lista doblemente ligada, es posible comenzar a buscar desde el final (para, por ejemplo, obtener el elemento 99 de una lista de 100 por medio de 2 iteraciones y no de 99), sigue siendo supremamente eficiente, por ejemplo, acceder al elemento 50. Esta ineficiencia se va volviendo cada vez más preocupante conforme crece el tamaño de la lista.

El problema aquí es que algoritmos de ordenamiento como QuickSort requieren acceder repetidamente a elementos por su índice, lo cual ralentiza enormemente el proceso de ordenamiento, especialmente en listas muy largas. Es por esto que, para efectos de ordenamiento de datos, las listas doblemente ligadas son altisimamente ineficientes.

Una ventaja de la lista doblemente ligada como fue previamente mencionado es que puede analizar una cantidad indefinida de nodos, a base de la cantidad de líneas leídas en un documento; esto le permite al programa tener la flexibilidad de trabajar con diferentes tipos de archivos y, mientras estén ordenados con la estructura predefinida, sea posible presentar

una solución que uno o varios individuos tardarían una cantidad exponencialmente grande de tiempo en producir y comprobar su veracidad. Sin embargo, esta ventaja será provechosa solo en el caso de que la información no se tenga que ordenar.

Reflexión:
En esta actividad se trabajó con una lista doblemente ligada y se utilizó uno de los ordenamientos para trabajar con ips. Fuera del contexto de esta actividad se consideró que esta estructura de datos es muy ineficiente o no es la apropiada para hacer un ordenamiento. Esto se debe a la flexibilidad pero también a la rigidez que tiene una lista entrelazada doble. Independiente a eso se pudo entender a profundidad cómo los apuntadores y trabajar cómo hacer las modificaciones al archivo 1.3 para poder correr este código de manera eficiente.

Reflexión 3.2-

La importancia y eficiencia del uso de BST en una situación problema de esta naturaleza

La importancia de un BST para poder tratar con una situación problema de esta naturaleza es que permite encontrar fenómenos y patrones por la misma naturaleza de la estructura de datos. En la bitácora que se trabajó se puede evidenciar las siguientes cinco y IPS con más acceso.

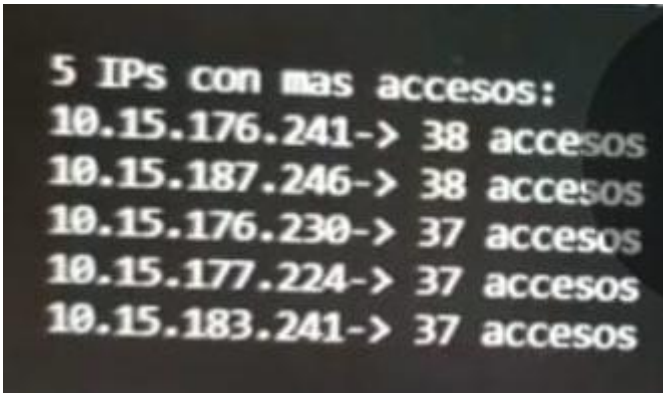


Imagen 1 - Las 5 IPs de la bitácora con más acceso

En la bitácora se puede hacer referencia a estos ipes y se puede evidenciar que las 5 y IPS con más accesos dan a usuarios ilegales que están tratando de entrar al sistema y estos son rechazados en los diferentes niveles de acceso.

```
Jun 2 22:24:15 10.15.173.220:4014 Illegal user
Jul 10 14:20:29 10.15.188.231:6314 Illegal user
Jun 25 15:31:01 10.15.185.252:5762 Failed password for illegal user guest
Oct 23 1:23:16 10.15.182.233:4640 Failed password for admin
Aug 21 19:32:57 10.15.170.247:4074 Failed password for illegal user guest
Jun 1 23:35:58 10.15.176.250:6009 Failed password for illegal user test
Sep 20 13:10:51 10.15.171.228:6300 Failed password for illegal user test
Jul 30 18:23:55 10.15.171.250:6353 Failed password for illegal user test
Oct 28 3:59:32 10.15.176.241:6550 Illegal user
Oct 11 11:26:43 10.15.184.237:5818 Failed password for illegal user guest
```

Imagen 2 - Primer IP con mayor acceso

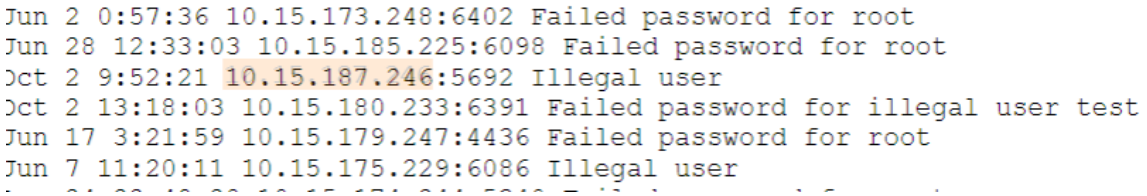


Imagen 3 - Segundo IP con mayor acceso

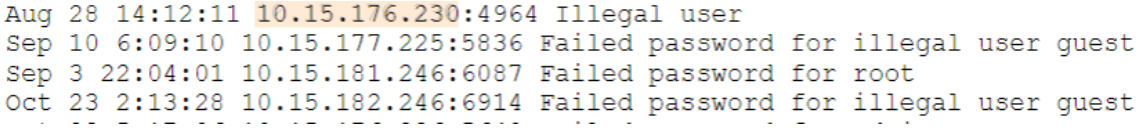


Imagen 4 - Tercero IP con mayor acceso

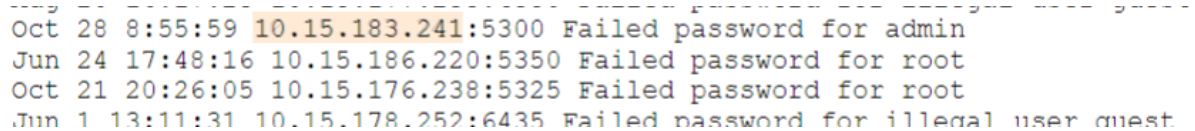
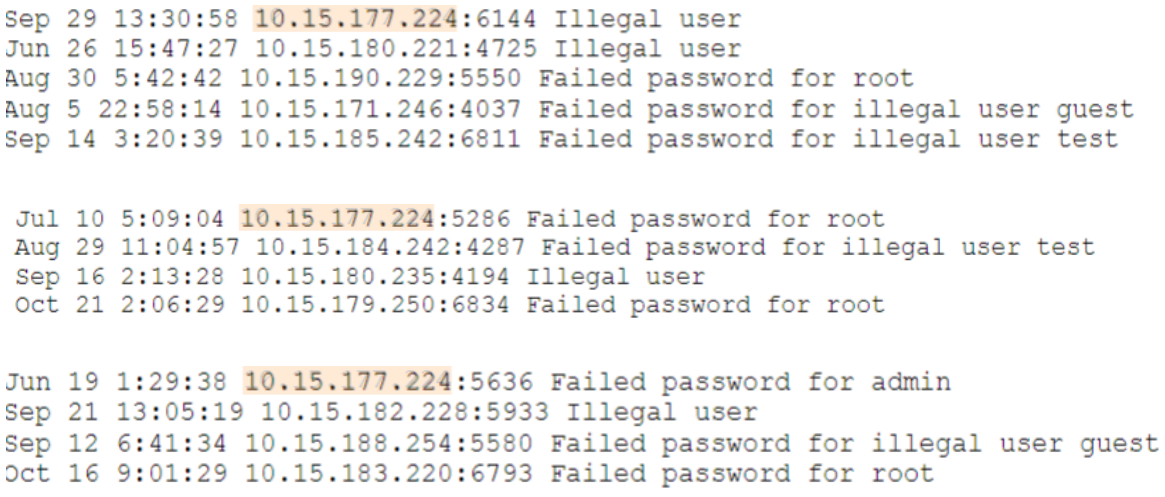


Imagen 4 - Cuarto IP con mayor acceso

En la IP 10.15.177.224 se trata ejemplificar a mayor profundidad este caso donde se puede evidenciar que en diferentes fechas el usuario trato de acceder al sistema dónde se lo detectó como usuario ilegal, trato y fracaso en poner una contraseña para el root y también fracasó en poner la contraseña para la cuenta de admin. En este caso hay 37 accesos y solo se ejemplifican tres pero cabe resaltar el comportamiento de los casos de ips más registrados en una bitácora dónde se registra la razón de porque no puedo entrar un usuario.



Se considera que el método BST además de ser eficiente e importante en una situación problema con una naturaleza delicada como la presentada en esta actividad, también por las reglas de ordenamiento de la misma estructura de datos permite encontrar posibles vulnerabilidades en el sistema.

Otro punto que tiene que ser tomado a reflexión es que estos datos sin la interpretación humana que investigue y realmente ponga criterio el significado y las consecuencias de usuarios como estos. La importancia o eficiencia del BST no serviría dado que esta es una herramienta para poder tomar decisiones más no el único método.

¿Cómo podrías determinar si una red está infectada o no?

Según estudios casi un tercio de todos los ordenadores en el mundo se reportan tener algún tipo de malware. Donde no necesariamente sea posible evidenciar un equipo que está infectado pero la manera principal de detectarlo es con software anti-malware o spyware o cualquier tipo de software malicioso. Estos programas buscan en todo el ordenador y están desarrollados de tal manera que saben dónde buscar y encontrar la gran cantidad de archivos ya que hacen referencia seguramente a una o miles de bibliotecas para poder identificar posibles rastros o malwares presentes en un ordenador.

Como referencia en Estados Unidos 82% de las casas conectadas al internet tienen algún tipo de software antier según security.org. De esta estadística corresponde a 86 millones de casa que reportan utilizar este tipo de software y casi la mitad de ellos pagan por un servicio para proteger sus dispositivos. Por lo tanto en varios países desarrollados se pudiera estimar que se está tomando conciencia y por ende se asumirá que también en todo el mundo ya que es un tema de índole universal. Se considera que fuera de hacer chequeos constantes y verificar que todo el sistema esté funcionando correctamente con alguno de estos softwares o servicios es difícil no encontrar algún tipo de malware en un ordenador.

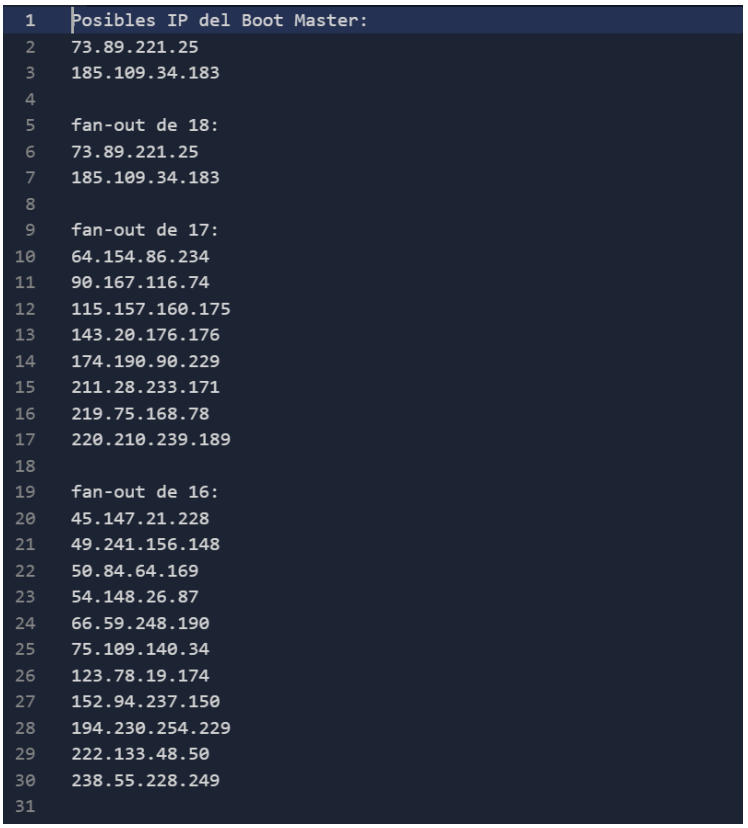
Reflexión 4.3-

El documento refleja la importancia y eficiencia del uso de grafos en una situación problema de esta naturaleza.

En esta actividad se trabajó con grafos por medio de una lista de adyacencia. Los grafos en de esta naturaleza son importantes dado que este tuvo una cantidad considerable de datos qué tenían que ser analizados. Dónde se requirió analizar más de 13000 diferentes IPs y 90000 artistas se trabajó con una lista de adyacencia particularmente útil dado que permite encontrar la presencia o ausencia de un borde específico entre dos nodos. De igual manera esta estructura de datos nos permitió iterar rápidamente sobre todas las aristas dado que nos permite acceder a cualquier vértice adyacente de manera directa. Por estas razones se considera a esta estructura de datos eficiente y útil para resolver este problema.

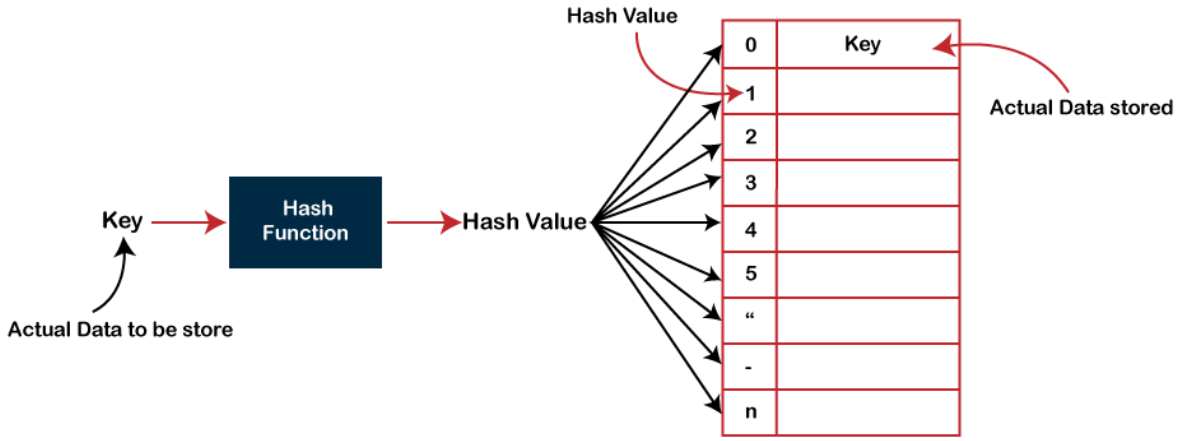
Para resolver las preguntas presentando el fan out en orden descendente, se puede encontrar los simples que son llamados con la mayor frecuencia. Al mismo tiempo se pide determinar el IP qué puede ser el boot master.

En la imagen inferior puede evidenciar los dos IPS dónde que se infiere que pueden ser los bootmasters.



Reflexión 5.2-

Los hash tables son una lista de valores pares donde el primer valor es una llave y el segundo es el valor, donde se puede acceder de manera rápida al valor por medio de la llave. Si se desea hacer búsquedas rápidas, esta estructura es una de las más eficientes y por sus propiedades también es útil para eliminar o insertar elementos con una complejidad de O(n) en el peor de los casos.



Como asigna una hashtable sus valores

Sin embargo esta estructura puede ser no la más eficiente si se requiere estructura dado que esta no mantiene un orden en específico, pero como fue analizado previamente para estas entregas si se considera ideal.

En el caso del trabajo realizado por el equipo se llegó a la misma solución utilizando unorder_map pero se consideró por motivos de aprendizaje y también de investigación más útil demostrar su uso desarrollando las tablas hash y no una librería de C++ que prácticamente hace todo el hashing sin hacer realmente la actividad.

Se puede considerar que la bitácora tiene 91000 entradas y no es posible que este tengo la misma cantidad de ips si estos se repiten varias veces y al mismo tiempo en los primeros dos octetos de cada ip hay una posibilidad de 256 combinaciones del 0-255 haciendo 65536 posibles combinaciones como se puede ver en la imagen inferior del código.

```cpp
#define TABLE_SIZE 65536 //256^2

using namespace std;

class IPHashTable{
    private:
        forward_list<IPSummary> ipSummaries[TABLE_SIZE];
        //Esta es la función hash, que con la IP (el key) determina el índice dentro del arreglo
        int getIPIndex(string ipAsString){
            int octets[2];
            for(int i = 0; i < 2; i++){
                int dotIndex = ipAsString.find('.');
                octets[i] = stoi(ipAsString.substr(0, dotIndex));
                ipAsString = ipAsString.substr(dotIndex+1);
            }

            return octets[0]*256 + octets[1];
        }
```

Para poder asignar cada ip a una “key” lo que se realizó en el ip es multiplicar el primer octeto por del ip por 256, el número de combinaciones y sumarle el valor del segundo octeto generando una llave particular que sería llamada cada vez que esta aparezca y donde se pueda guardar los otros datos en la lista que se creó.

En los hash tables se tiene que considerar que no es posible saber de antemano cuántas colisiones pueden haber con una función hash. Sin embargo, prácticamente todas las implementaciones de tablas hash ofrecen O(1) en la gran mayoría de insertos. Esto es lo mismo que insertar matriz: es O(1) a menos que necesite cambiar el tamaño, en cuyo caso es O (n), más la incertidumbre de colisión. Donde para prácticamente todos los casos de uso, las tablas hash son se tiene una complejidad O(1) donde la complejidad de inserción es O(1) y la de búsqueda es O(1).

Como fue previamente mencionado, fuera del caso que se tenga dos datos en una memoria que es cabe resaltar que es bastante raro, la complejidad de cada operación es O(n) y por lo tanto esta estructura de datos es altamente eficiente en términos de complejidad y utilidad dado que es una estructura de datos ideal para resolver problemas de esta índole.

En la parte inferior se puede ver como se encontró la cantidad de veces que fue llamado un IP arbitrario en la entrega que en este caso sería la llave y los datos que aparecen como las fechas y razones de la falla.

```
Escribe la direccion IP de la que quieres el resumen (escribe -1 para terminar): 195.44.205.157

Accesos a la IP 195.44.205.157 (8 accesos):
Puerto: 4169 | Fecha: Jul 09 16:19:58 | Razon de falla: Illegal user
Puerto: 3742 | Fecha: Jun 09 01:32:22 | Razon de falla: Failed password for illegal user guest
Puerto: 6129 | Fecha: Jul 02 16:36:33 | Razon de falla: Failed password for admin
Puerto: 3355 | Fecha: Oct 13 20:44:49 | Razon de falla: Failed password for illegal user guest
Puerto: 9040 | Fecha: Jun 10 03:20:15 | Razon de falla: Failed password for admin
Puerto: 3098 | Fecha: Jul 26 04:39:18 | Razon de falla: Failed password for illegal user guest
Puerto: 4581 | Fecha: Aug 26 04:22:30 | Razon de falla: Failed password for illegal user guest
Puerto: 4021 | Fecha: Sep 25 18:11:34 | Razon de falla: Failed password for illegal user root
```

Reflexión 6.2-

¿Cuáles son las más eficientes?

Para las actividades realizadas se considera que con todos los puntos previamente mencionados las estructuras más eficientes fueron los BST dado su complejidad computacional y su uso particular la actividad 3.4 y los hash tables en la actividad 5.2. Se consideró que para algunas de las actividades como la 1.3 se uso el metodo quicksort junto al método de búsqueda binaria para encontrar todos las fechas que corresponden a las fechas pedidas de inicio a fin, sin embargo esto fue utilizado como alternativa a usar una estructura de datos en particular y se tuvo que crear una estructura Bitácora para poder utilizar los métodos.

De igual manera en la 2.3 se deseaba realizar un double linked list para poder encontrar todos los IPs de fechas particulares y este no era eficiente, dado que requería buscar los datos de manera innecesaria y se consideró que era una estructura de datos errónea para los requerimientos o por lo menos no la más eficiente. Pero se considera que los hash tables para casi todas las actividades era el método más eficiente dado que era el más fácil de implementar y aunque se tiene que considerar que no es posible saber de antemano cuántas colisiones pueden haber con una función hash. Sin embargo, prácticamente todas las implementaciones de tablas hash ofrecen O(1) en la gran mayoría de insertos. Es O(1) a menos que necesite cambiar el tamaño, en cuyo caso es O (n), más la incertidumbre de colisión. Donde para prácticamente todos los casos de uso, las tablas hash son se tiene una complejidad O(1) donde la complejidad de inserción es O(1) y la de búsqueda es O(1). En base a lo presentado, de todas las evidencias se considera que la última estructura de datos estudiada es la más eficiente para buscar aunque tenga complejidades iguales a otras estructuras.

En términos de eficiencia, realmente depende del caso particular que se está tratando para poder determinar cuál estructura de datos es la apropiada para tratar ese caso en específico. Una estructura de datos puede ser ideal en ciertas situaciones pero en otras es subóptima por eso se justifica realmente el uso y creación de diferentes estructuras de datos. No hay una estructura que cumpla con todas la características y a base de esta premisa se puede comparar la eficiencia de todas las estructuras de datos estudiadas en este curso.

Esto lleva a la consideración que las estructuras de datos permiten almacenar y representar elementos de una cierta manera, pero este componente también debe ser considerado como una parte del sistema y no es más de una herramienta. Por lo tanto si la pregunta es cuales son los más eficientes como se mencionó previamente esto depende de la necesidades que se aplican, pero si se tiene un caso donde dos o más estructuras de datos son “igual de eficientes” se debe buscar la estructura que maximice todos los requerimientos y casos de uso. A continuación se hace un análisis de las principales estructuras estudiadas en el curso donde se determina su uso, complejidad, y eficiencia dependiendo del caso.

Un arreglo es la estructura más básica de datos donde su complejidad es siempre O(n) y esta es la manera más fácil de representar una lista de elementos.

Una doble lista ligada en su caso promedio es O(n log n) y el peor caso es O(n^2), donde se usaría esta estructura si se requeriría eliminar o insertar algún elemento al comienzo de la lista ligada. El linkedlist es similar a un arreglo donde se representa una lista de arreglos, pero este maneja su memoria diferentemente al esparcir sus unidades de memoria. En el peor caso un linkedlist doble puede ser menos eficiente que un arreglo pero en el caso promedio este tiendo a ser mejor por lo tanto en este caso sería "más" eficiente.

Los hash tables son una lista de valores pares donde el primer valor es una llave y el segundo es el valor, donde se puede acceder de manera rápida al valor por medio de la llave. Si se desea hacer búsquedas rápidas, esta estructura es una de las más eficientes y por sus propiedades también es útil para eliminar o insertar elementos con una complejidad de O(n) en el peor de los casos. Sin embargo esta estructura puede ser no la más eficiente si se requiere estructura dado que esta no mantiene un orden en específico, pero como fue analizado previamente para estas entregas si se considera ideal.

En el caso de filas o columnas(queues), estas son eficientes como contenedores temporales para poder mover datos sin embargo esta tiene limitaciones, una fila entraría el primer dato en entrar es el primero en salir, en cambio en una queue el primer dato en entrar es el último en salir donde cada elemento es insertado con una complejidad O(1) y la complejidad de la estructura es O(n).

En el caso de un árbol binario su complejidad es O(n) que tiene una regla principal que es que un nodo padre no puede tener más de dos nodos hijos y este es eficiente en el caso de que se quiera comparar los elementos de una manera tal que se evalúe si son menor o mayor. Sin embargo en el caso del uso de recursión, un árbol binario se volvería poco eficiente. En comparación los árboles de búsqueda binarios que es una variante de esta, esta estructura tiene una complejidad promedio de O(log n) con base 2 y de O(n) en el peor de los casos por lo tanto este si es una estructura de datos más eficiente en comparación a los árboles binarios simples dado que estos remueven medio subárbol cada paso que justifica su complejidad.

En el caso de grafos está es un colección de nodos con artistas y dependiendo de su estructura de datos sea una matriz adyacente con complejidad O(V^2) o lista adyacente O(V+E), su eficiencia dependerá de lo que se necesita pero este puede ser el más eficiente si se desea mostrar la conexión o relación entre diversos datos.

¿Cuáles podrías mejorar y argumenta cómo harías esta mejora?

Todos pudieran mejorar desde un punto de vista técnico,

Para la actividad 1.3, se utilizó quicksort como algoritmo de ordenamiento pensando que este es mejor en tiempo real que mergesort y que su complejidad promedio es O(nlogn), sin embargo su complejidad en el peor caso es O(n^2) mientras que la complejidad de tiempo en el peor caso de mergesort es O(nlogn). Además después de la implementación y un poco de investigación en la literatura encontramos que quicksort tiende a ser un mejor método de ordenamiento para datasets o en este caso bitácoras pequeñas y que mergesort por su complejidad es mejor para archivos grandes. Con todo esto a consideración para poder mejorar esta entrega se implementa mergesort que reduciría la complejidad del código en cuestión de tiempo, tomando a consideración que quicksort tiene una complejidad de espacio de O(logn) y megesort de O(n) donde espacio no es algo que se está buscando optimizar.

Como se destacó en la reflexión 2.3 y en la pregunta previa donde se trabajó con una lista doblemente ligada y se utilizó uno de los ordenamientos para trabajar con ips. Fuera del contexto de la actividad 2.3 se consideró que esta estructura de datos es muy ineficiente o no es la apropiada para hacer un ordenamiento. Esto se debe a la flexibilidad pero también a la rigidez que tiene una lista entrelazada doble .Otra desventaja de las listas ligadas es el tiempo que toma acceder a uno de sus elementos por índice. Para, por ejemplo, acceder al quinto elemento de la lista, se tiene que iterar 5 veces, accediendo a los 4 nodos anteriores al quinto. Por lo tanto se consideró que si se aplicaba us hashtable o un grafo en vez de un doble lista ligada se hubiera hecho el código de manera más simple y eficiente en un menor tiempo.