Jaime Eduardo López Castro - A00833173

Introducción:

“En ciencias de la computación, una estructura de datos es una forma particular de organizar datos en una computadora para que puedan ser utilizados de manera eficiente.”

De manera simple, la memoria de una computadora es lineal, una tira gigante de espacios puestos uno tras del otro, esperando a ser accedidos y modificados; sin embargo, no tenemos por qué limitar las instrucciones que damos a esta línea, pensar fuera de la caja, manipulando esta línea de diferentes formas puede crear diferentes formas de almacenar datos.

Durante las actividades realizadas este verano experimentamos acerca de esto justamente, utilizamos diferentes estructuras de datos y algoritmos para crear programas eficientes. Como resultado hicimos un código que lee un poco más de 100,000 inputs, almacena cada elemento junto con las relaciones entre ellos y es capaz de imprimir todas las conexiones salientes de cada elemento en orden descendente, todo en un tiempo promedio de medio segundo.

Jaime Eduardo López Castro - A00833173

Reflexión - Actividad 1.3:

Lo que nuestro programa de la actividad 1.3 hace es básicamente que lee los registros de una bitácora creando un objeto de tipo Registro para cada línea y guardándolo en un vector dentro de una clase Bitácora, donde podemos modificarlo después.

Si queremos buscar un elemento dentro de esta bitácora, es más eficiente ordenarla de acuerdo a cierto atributo y luego utilizar una búsqueda binaria para encontrarlo más rápidamente.

En nuestro caso utilizamos MergeSort para ordenar los elementos en el vector con respecto a su Unix Time, en general no es definitivamente mejor que QuickSort, pero su tiempo de ejecución es ligeramente más rápido, es por eso que utilizamos este algoritmo.

Tabla: Complejidad de los algoritmos de ordenamiento.

Algoritmo	Mejor	Promedio	Peor	Estable	Espacio
Swap sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Una vez ya tenemos ordenado nuestro vector, es cuestión de utilizar Búsqueda Binaria para dar recorrer el vector en O(log n); como implementamos sobrecarga de operadores en la clase Registro, no hay ningún inconveniente con usar el código que hicimos en actividades anteriores.

Tabla: Complejidad de los algoritmos de búsqueda.

Algoritmo	Mejor	Promedio	Peor
Búsqueda secuencial	$O(1)$	$O(n)$	$O(n)$
Búsqueda secuencial en arreglos ordenados	$O(1)$	$O(n)$	$O(n/2)$
Búsqueda binaria	$O(1)$	$O(\log_2 n)$	$O(\log n)$

Así es como al final, después de haberlo ordenado y luego buscado por búsqueda binaria, obtenemos una complejidad de O(n log n + log n) = O(n log n), mucho mejor que si la hubiéramos buscado elemento por elemento, con complejidad de O(n).

Jaime Eduardo López Castro - A00833173

Reflexión - Actividad 2.3:

En esta implementación utilizamos una lista doblemente ligada en vez de una sencilla, esto tiene la ventaja de que podemos recorrer el contenido de nuestra bitácora al revés en vez con complejidad lineal en vez de cuadrática, como sería con una lista simple. La única desventaja que presenta esto es que una lista doblemente enlazada utiliza más memoria ya que cada nodo tiene como atributo también un apuntador a su nodo previo.

La complejidad final de nuestra implementación es de O(log n) para la búsqueda de un elemento después de ordenar la bitácora ya que utilizamos el algoritmos de búsqueda binaria y O(n log n) para ordenar la bitácora utilizando nuestra implementación de MergeSort. Las complejidades de la lectura e impresión son lineales, ya que tenemos que visitar todos los nodos existentes en nuestra lista.

Siguiendo con la comparación entre una lista ligada y una lista doblemente ligada; las complejidades en sus operaciones básicas es la misma: O(1) para insertar y eliminar elementos al principio y al final gracias a los apuntadores de head y tail, O(logn) para buscar un elemento con la bitácora ordenada, como ya había mencionado y O(n) para buscarlo en la lista desordenada ya que estas no funcionan por indexado como los vectores o los arrays; O(n) para insertar o eliminar elementos a puntos intermedios de la lista, ya que tenemos que recorrerla toda elemento por elemento para poder llegar a la posición deseada. En nuestro caso no vamos a insertar ni borrar elementos entre medio de la lista, por lo que sólamente nos interesa saber su complejidad para insertar, borrar y ordenar la bitácora.

En cuanto al algoritmo de ordenamiento que utilizamos, implementamos los 2 y los comparamos en tiempo promediando 5 tests para así poder saber cuál resulta mejor en nuestro caso.

Jaime Eduardo López Castro - A00833173

QuickSort: promedio user - 0.214 ms

```
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffd821c2060
--- Liberando memoria de la lista doblemente ligada --- 0x7ffd821c2060

real    0m0.337s
user    0m0.211s
sys 0m0.050s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffc08e76300
--- Liberando memoria de la lista doblemente ligada --- 0x7ffc08e76300

real    0m0.373s
user    0m0.196s
sys 0m0.052s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffd37af6a20
--- Liberando memoria de la lista doblemente ligada --- 0x7ffd37af6a20

real    0m0.300s
user    0m0.225s
sys 0m0.031s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffe2dc7e410
^[[A--- Liberando memoria de la lista doblemente ligada --- 0x7ffe2dc7e41

real    0m0.956s
user    0m0.253s
sys 0m0.049s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffd61386150
--- Liberando memoria de la lista doblemente ligada --- 0x7ffd61386150

real    0m0.575s
user    0m0.204s
sys 0m0.050s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffd2fe735a0
--- Liberando memoria de la lista doblemente ligada --- 0x7ffd2fe735a0

real    0m0.323s
user    0m0.195s
sys 0m0.061s
```

Jaime Eduardo López Castro - A00833173

MergeSort: promedio user - 0.0702 ms

```
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffc0957d8b0
--- Liberando memoria de la lista doblemente ligada --- 0x7ffc0957d8b0

real    0m0.304s
user    0m0.089s
sys 0m0.040s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7fffd63d6220
--- Liberando memoria de la lista doblemente ligada --- 0x7fffd63d6220

real    0m0.145s
user    0m0.050s
sys 0m0.076s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7fffea4bcbc0
--- Liberando memoria de la lista doblemente ligada --- 0x7fffea4bcbc0

real    0m0.112s
user    0m0.060s
sys 0m0.045s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7ffe474c9fe0
--- Liberando memoria de la lista doblemente ligada --- 0x7ffe474c9fe0

real    0m0.138s
user    0m0.060s
sys 0m0.056s
time ./main

--- Creando una lista doblemente ligada vacia --- 0x7fff64d32f30
--- Liberando memoria de la lista doblemente ligada --- 0x7fff64d32f30

real    0m0.156s
user    0m0.092s
sys 0m0.036s
```

Reflexión - Actividad 3.4:

Para esta actividad utilizamos una fila de prioridad que se basa físicamente en un vector para poder acceder a los elementos mediante índices. Esto tiene la ventaja de que podemos extraer, en este caso, el elemento más grande del conjunto en tiempo O(1) ya que se encuentra al principio del vector; sin embargo, como al insertar o borrar un elemento se tiene que reacomodar el vector, la complejidad de estas operaciones es O(log n), que sigue siendo muy rápido, sin embargo, no tanto como hacerlo en una lista enlazada, a pesar de esto, para nuestro objetivo es un sacrificio que vale la pena hacer.

La complejidad final del programa es de O(nlog n) para leer el txt de los datos, O(nlog n) para ordenar los registros por ip, O(n) para exportar estos registros a un txt, O(nlog n) para contabilizar las repeticiones por ip, O(nlog n) para volver a ordenar todas las ip por número de repeticiones, O(n) para exportar estas ips a un txt y por último O(1) para obtener los 5 registros más grandes de esta lista, ya que ya tenemos armado el vector; por lo que tenemos una complejidad final de O(4*nlogn + 2*n + 1) = O(nlogn), que se mantiene igual si no tomamos en cuenta la creación de los txt.

Gran parte de por qué es tan buena la eficiencia de este programa es por el uso de la lista de prioridad, que se comporta en parte como un BST, ya que su búsqueda es O(log n) y la complejidad de la búsqueda del elemento más grande es constante. Si lo comparamos con un vector simple, la búsqueda es O(n) y por si no fuera poco, para encontrar el elemento más grande la complejidad también es O(n), por lo que si quisiéramos hacer HeapSort de esta manera obtendríamos una complejidad de O(n^2), que tardaría mucho tiempo con los 16,808 registros que tenemos en nuestro txt.

```
 ---> Creando un MaxHeap por default: 0x7ffc4f15f360
 ---> Creando un MaxHeap por default: 0x7ffc4f15f380

    IP          Repeticiones
10.15.187.246       38
10.15.176.241       38
10.15.183.241       37
10.15.176.230       37
10.15.177.224       37

 ---> Liberando la memoria del MaxHeap: 0x7ffc4f15f380
 ---> Liberando la memoria del MaxHeap: 0x7ffc4f15f360
```

Jaime Eduardo López Castro - A00833173

Reflexión - Actividad 4.3:

En esta implementación utilizamos 4 estructuras de datos destacables: Un grafo ponderado dirigido, listas enlazadas, maps de la implementación del STL y MaxHeaps (listas de prioridad). Utilizamos el grafo por su funcionalidad, ya que no hay mejor estructura para representar diferentes elementos interconectados entre sí, en este caso direcciones de Ip que realizan ataques cibernéticos a otras. En la implementación del grafo, utilizamos una lista enlazada simple, ya que la forma en la que almacenamos a este es mediante una lista de adyacencia, esto porque es más fácil extender las conexión entre nodos mientras vamos leyendo nuestro txt, gracias a que la complejidad de añadir elementos a una lista enlazada es O(1). Luego utilizamos el map, que está implementado en una estructura de tipo árbol binario de búsqueda que se ordena en base a un índice, lo que utilizamos para buscar la posición de ciertas ips en el grafo, la complejidad de búsqueda de un map es O(logn). Por último tenemos al MaxHeap, que utilizamos para el algoritmo de HeapSort, que tiene una complejidad de O(nlogn) y además nos sirve para obtener los valores máximos de un conjunto de datos en complejidad O(1).

La complejidad final del programa es O(V * E * logn), número de vértices por el número de enlaces por log de n. Leer el txt inicial y crear un grafo en base a él tiene complejidad de O(n * V), el número de líneas multiplicado por el número de nodos; ya que hacemos una operación por cada línea, cuando estamos añadiendo nodos creamos una lista enlazada vacía por cada uno y la ponemos en el vector de la lista de adyacencia. Cuando exportamos al txt las ip con sus grados la complejidad es O(n), ya que tenemos que recorrer todo el vector de datos. Para encontrar el Boot Master tardamos O(logn) ya que buscamos un valor dentro de nuestro map. En la función de shortestPath tenemos una complejidad de O(V * E * logn), número de vértices por el número de enlaces por log de n, ya que recorremos todo el número de vértices, recorremos sus vecinos y comparamos sus distancias, por lo que en el peor de los casos también recorremos todos los elementos del grafo, después añadiendolo a un MaxHeap con complejidad de O(logn), esto para que después sea más fácil encontrar el valor con más conexiones de todos. En la función TopFive tardamos O(logn) ya que tenemos que sacar 5 elementos del heap y luego reacomodarlo. Por último para encontrar el nodo con la mayor distancia al Boot Master tardamos un tiempo constante, por lo que es solamente acceder al top de nuestro MaxHeap, esto es extremadamente rápido.

La mayor aportación a la eficiencia de nuestro programa es el uso de maps y MaxHeaps para buscar elementos entre el universo y para obtener el elemento más grande del conjunto. La complejidad de búsqueda en un vector es O(n), mientras que en un map es O(logn), una grandísima mejora; de otra forma, la mayor diferencia es que en un MaxHeap tardamos un tiempo constante en obtener el valor más grande, mientras que en un vector la complejidad es lineal, una diferencia abismal.

Jaime Eduardo López Castro - A00833173

Reflexión - Actividad 5.2:

La estructura de datos en la que se basó esta entrega fue en las HashTables, en nuestro caso específico, el uso de una HashTable de dirección abierta con prueba cuadrática, lo que significa que no utiliza memoria externa para tratar con las colisiones. Gracias a la función Hash implementada, añadir elementos tiene una complejidad de O(1), así como también buscarlos después, algo que no habíamos conseguido desde que empezó el curso y una gran mejora por sobre cualquier otra estructura de datos. Esta estructura funciona gracias a una llave numérica, que es la que entra a la función Hash, gracias a que utilizamos la IP numérica como llave, podemos almacenar en el nodo de la tabla el resto de datos de la IP, como su representación en string, conexiones de entrada y salida, entre otras cosas. En cuanto al tamaño inicial de esta estructura, decidimos utilizar un valor de 26759, que es un número primo que representa un poco más del doble de elementos que contamos en nuestra bitácora, nos dimos cuenta que utilizando este método logramos reducir las colisiones a 2,445; a diferencia de cuando utilizamos los espacios justos que obtuvimos 307,230 colisiones.

La complejidad final de nuestro programa fue de O(n * V) (el número de líneas en el txt por el número de vértices del grafo); esto es gracias a que leer el txt y crear un grafo en base a él tiene esta complejidad, gracias a que tenemos que crear un vector de listas enlazadas para representar nuestra lista de adyacencia, posteriormente tenemos que crear una lista enlazada vacía y añadirla a nuestro vector de adyacencias por cada nodo, terminando con leer cada uno de los enlaces y ponerlo en el nuestra lista de adyacencias.

Justamente gracias a que añadir elementos a nuestra HashTable tiene un tiempo en promedio constante, poblar nuestra tabla tiene una complejidad de O(n), ya que tenemos que poner elemento por elemento. Ya una vez cargada nuestra tabla en cuando empieza la magia, buscar elementos y obtener sus datos es en promedio O(1), lo único que nos toma tiempo es imprimir las conexiones de esta ip específicamente de forma ascendente, ya que para esto creamos un Heap, lo llenamos de todas las conexiones de salida de nuestra ip y realizamos un HeapSort para imprimirlas en la pantalla, tomándonos un tiempo de O(nlogn).

Intentar realizar esta evidencia con alguna otra estructura que no fuera una HashTable tendría consecuencias en su tiempo de ejecución, un vector por ejemplo tardaría un tiempo O(n) en buscar un elemento si están desordenados y si quisiéramos ordenarlos para realiza un algoritmo más eficiente, este no sería mejor a O(nlogn). Una estructura del estilo de un BST es más rápida que la búsqueda lineal, pero sigue sin ser constante, por lo que utilizar una HashTable es sin duda nuestra mejor opción aquí.

Conclusión:

Después de todas estas reflexiones, no podemos hacer otra cosa más que concluir que saber acerca de estructuras de datos y algoritmos es increíblemente importante para poder realizar programas eficientes, fáciles de leer, concisos y expandibles.

Una de las primeras cosas que se nos enseña al momento de aprender a programar es a utilizar vectores (o arreglos), ya que son una estructura simple, eficaz y muy útil; sin embargo, esta no es la mejor opción para todos nuestros problemas. Si, por ejemplo, pusiéramos un montón de elementos en nuestro vector y quisiéramos encontrar la posición de uno en específico, en el peor de los casos tendríamos que recorrer todo el vector elemento por elemento para poder dar una respuesta; esto no es un gran problema para 10 o 15 datos, pero cuando tratamos con volúmenes que se elevan a los millones, encontrar tanto una buena estructura de datos como un buen algoritmo es la diferencia entre un programa que tarda 500 años en ejecutarse, a otro que corre en unos pocos segundos.

No tenemos por qué conformarnos con la primera forma de almacenar información que se nos enseñó, una computadora es la herramienta más poderosa que podemos poseer, sus datos pueden ordenarse y relacionarse de formas muy abstractas, que hacen que a ojos de cualquiera, nuestras pocas líneas de código hagan magia. El mundo funciona a través de formas eficientes e inteligentes de leer, procesar y almacenar datos, así que entender los métodos por los cuales esto se lleva a cabo es la mejor forma de aportar al desarrollo de la sociedad.

Volviendo a nuestro caso específico, nos dimos cuenta que las HashTables son estructuras extremadamente rápidas cuando hablamos de buscar y añadir elementos, todo dependiendo de su función Hash y el tamaño del vector en el que se almacenen los datos; a pesar de que esto es increíble, esta estructura tiene la desventaja de que no hay algoritmos especialmente eficientes para ordenar los datos almacenados, ahí es en donde entra una estructura secundaria, en nuestro caso fue un MaxHeap, ya que con este tenemos la ventaja de poder acceder al elemento más grande del conjunto muy rápido (ya que siempre está en la primera posición) y podríamos ordenar nuestra información utilizando el algoritmos de HeapSort si así lo deseáramos. La única cosa que se me ocurre que podría mejorar la eficacia de nuestro producto final es el crear una mejor función Hash que nos ayudara a reducir las colisiones generadas, así como a reducir el tamaño utilizado por nuestro vector.

Jaime Eduardo López Castro - A00833173

Fuentes:

- GeeksforGeeks. (2020, 1 diciembre). Advantages, Disadvantages, and uses of Doubly Linked List. https://www.geeksforgeeks.org/advantages-disadvantages-and-uses-of-doubly-linked-list/
- GeeksforGeeks. (2022, 3 julio). Merge Sort for Doubly Linked List. https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/
- GeeksforGeeks. (2022a, junio 24). Binary Search on Singly Linked List. https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/
- Kiao, U., PhD. (2022, 20 marzo). Time Complexity Analysis of Linked List. OpenGenus IQ: Computing Expertise & Legacy. https://iq.opengenus.org/time-complexity-of-linked-list/
- GeeksforGeeks. (2022a, mayo 7). Graph and its representations. https://www.geeksforgeeks.org/graph-and-its-representations/
- GeeksforGeeks. (2022d, julio 6). map find() function in C++ STL. https://www.geeksforgeeks.org/map-find-function-in-c-stl/#:%7E:text=The%20time%20complexity%20for%20searching,searching%20is%20O(1).
- Hash table runtime complexity (insert, search and delete). (2012, 9 febrero). Stack Overflow. https://stackoverflow.com/questions/9214353/hash-table-runtime-complexity-insert-search-and-delete#:%7E:text=Hash%20tables%20suffer%20from%20O,each%20element%20to%20the%20table%5D.
- GeeksforGeeks. (2022e, julio 17). Hashing | Set 1 (Introduction). https://www.geeksforgeeks.org/hashing-set-1-introduction/
- GeeksforGeeks. (s. f.). Hashing Data Structure. https://www.geeksforgeeks.org/hashing-data-structure/