

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

TC1031

Reflexión

“Actividad 4.3”



Alumna:

Sarah G. Martínez Navarro

A01703113

Fecha:

17 / Junio / 2022

Profesor:

Eduardo Arturo Rodríguez Tello

ÍNDICE

Tabla de contenidos	2
Introducción	3
Actividad 1.3	3
a) Estructura del código	3
b) Algoritmo de ordenamiento	4
c) Algoritmo de búsqueda	4
d) Conclusión	5
e) Referencias	6
Actividad 2.3	7
a) Estructura del código	7
b) Listas Enlazadas	9
c) Conclusión	10
d) Referencias	11
Actividad 3.4	12
a) Estructura del código	12
b) Heap y MaxHeap	15
c) ¿Cómo saber si una red está infectada o no?	16
d) Conclusión	16
e) Referencias	17
Actividad 4.3	18
a) Estructura del código	18
b) Grafos	19
c) Conclusión	20
d) Referencias	20

Introducción

¡El internet se encuentra bajo ataque! Con el avance de la tecnología, también han avanzado los ataques cibernéticos de los diferentes bots que buscan datos personales, información bancaria, entre otros datos que confiamos que deben estar seguros dentro de nuestros dispositivos. Sin embargo, las amenazas de ciberseguridad se encuentran ahora más presentes que antes y toda nuestra información está bajo el peligro de robo. En este reto debemos filtrar e identificar los posibles accesos maliciosos a través de los diferentes algoritmos que aprenderemos a lo largo de nuestra materia “Programación de estructuras de datos y algoritmos fundamentales”. Para desarrollar un programa que identifique exitosa y eficientemente estos accesos tendremos que tener un conocimiento profundo de las posibilidades que tenemos para programar para realmente elegir los algoritmos que nos entregarán los resultados correctos en el menor tiempo posible. Este proyecto estará dividido en múltiples secciones y actividades para llevar a cabo el constante desarrollo del programa que entregaremos al final del semestre. Con esto, pasamos al análisis de las actividades que conformarán el resultado final.

Actividad 1.3 (Búsqueda y Ordenamiento)

a) Estructura del Código

En nuestra primera actividad, tuvimos que darle forma al programa que íbamos a desarrollar, la base de la pirámide que será nuestra entrega final. Para esto, primero se crearon los documentos que almacenan las diferentes fases del código para mantener la limpieza y poder acceder fácilmente a las diferentes partes del mismo.

En esta versión del código se encuentran 3 áreas principales:

- Main: El main consiste sólo de un documento con terminación *cpp* donde llamamos las funciones que desencadenan todos los procesos que tenemos desarrollados en nuestros otros documentos. No debe ser muy largo porque la información puede empezar a esconderse por ahí y empieza a perder coherencia. Se convertiría en lo que llamamos “código espagueti”. Se nos ha enseñado en algunas otras materias que la meta es tener un “código lasagna”, que quiere decir, en capas. Y de ahí, se desarrollan las siguientes secciones de código.
- Registro: La sección registro contiene 2 documentos, uno con terminación *cpp* y otro con terminación *h*. En el documento *Registro.h* declaramos lo que son las clases en las que vamos a guardar las funciones de la sección. Podemos declarar las funciones como públicas o privadas, dependiendo de si queremos que se vean en otras clases o si queremos mantener la limpieza del sistema. En *Registro.cpp* desarrollamos las funciones que se declararon en el documento *h* para que sean funcionales para el programa que estamos desarrollando. Esta sección se encarga de separar y analizar los registros individuales dentro de la bitácora que ingresamos como datos. Para esto, definimos el constructor de la clase Registro para partir el registro en partes para poder ordenar después la lista. Un constructor Registro contiene mes, día, hora, minutos, ip, puerto y razón de falla del acceso.
- Bitácora: Similar a la sección registro, la sección bitácora también se compone de un documento *cpp* y uno *h* pero mantiene un objetivo diferente a la sección registro, ya que el área bitácora se encarga de la bitácora ya

completa para organizar todos los registros ya que estos han pasado por la sección registro. Dentro de bitácora es donde también declaramos el vector ListaRegistros para guardar los datos (constructor) de registro.

b) Algoritmo de Ordenamiento

Tras tener ya nuestros datos divididos en registros dentro de bitácora, podemos empezar a trabajar con ellos. Antes de intentar buscar un dato en la lista enorme de registros que tenemos es necesario ordenarlos por las fechas del suceso para facilitar el proceso de búsqueda. En la clase, vimos múltiples tipos de algoritmos para ordenamiento, pero los 2 que más consideré para esta actividad fueron el Bubble Sort y el Quick Sort. Estos son los algoritmos más eficientes de su respectiva área, el Bubble Sort siendo más fácil de implementar que el Quick Sort.

El Bubble Sort es el más simple de los algoritmos de ordenamiento. Funciona creando una burbuja alrededor de uno de los elementos de la lista que está ordenando (empieza seleccionando el primer número) y cuando encuentra un número menor la burbuja ahora selecciona ese número, continuando este proceso hasta atravesar todo el arreglo. Una vez que termina, pasa el número seleccionado (que es el que ha definido es el más pequeño de todo el arreglo) hasta adelante y repite el mismo proceso hasta que el arreglo está completamente ordenado. Así de tedioso como suena, suele ser.

Este algoritmo tiene una complejidad de tiempo promedio de $O(n^2)$ que es bastante alto, pero hace sentido comparando con la cantidad de veces que recorre el mismo arreglo para ordenarlo.

Mientras, tenemos también el Quick Sort. Como lo dice su nombre, suele ser mucho más rápido y, por lo tanto, más eficiente que el Bubble Sort. Sin embargo, es un poco más complicado de implementar. El Quick Sort es un algoritmo que cae en la categoría de "Divide y Vencerás". Este proceso toma un número pivote, que es la mediana del listado y coloca los números menores a la izquierda y los mayores a la derecha del número pivote. Este procedimiento se repite de ambos lados de la lista y así sucesivamente hasta terminar de ordenar, pero como empieza a ordenar de ambos lados, el proceso se repite menos veces que el Bubble Sort.

El algoritmo de Quick Sort tiene una complejidad de tiempo de $O(n \log n)$ que es mucho más eficiente de lo que es el Bubble Sort.

Sin embargo, no se iba a definir hasta haberlo probado por lo que, con ayuda del profesor, implementamos ambos algoritmos y los probamos tanto con la bitácora acortada como con la bitácora completa. Los resultados fueron definitivos. El algoritmo Quick Sort

c) Algoritmo de Búsqueda

Tenemos 2 algoritmos muy sencillos de entender que estudiamos en esta materia para la búsqueda de elementos en un arreglo o, en este caso, un vector. Tenemos lo que son la búsqueda lineal y la búsqueda binaria.

La búsqueda lineal es cuando, por ejemplo, tenemos un arreglo y estamos buscando un valor pasando el índice uno por uno. El proceso empieza desde el índice **0** (el primer valor del listado) y se va moviendo a través de un *for* que determina “mientras que el valor del índice sea menor a la longitud del arreglo, se recorre el arreglo hasta encontrar el valor que buscamos”. La complejidad del algoritmo de búsqueda lineal es de $O(n)$. No tiene que recorrer el arreglo (o en este caso, la listaRegistros) múltiples veces, por lo que no es un procedimiento muy complejo, sin embargo, debido a que tiene que ir espacio por espacio para encontrar el valor ingresado, es más tardado que la búsqueda binaria y, por lo tanto, menos efectivo.

Por el otro lado tenemos lo que es la búsqueda binaria, que fue el algoritmo utilizado en este reto. La búsqueda binaria consiste de un proceso de “Divide y Vencerás” (similar a lo visto con el Quick Sort). El primer paso es definir el medio del arreglo (en este caso, la mediana del arreglo ordenado) que será nuestro punto para filtrar rápidamente nuestro arreglo. Una vez con la mediana (que por ahora llamaremos el valor ‘x’), tenemos, claro, 2 lados en el arreglo: los valores mayores a ‘x’ y los valores menores a ‘x’. Digamos que estamos buscando un valor ‘y’ en nuestro arreglo. Lo que hace el proceso es definir si nuestro valor ‘y’ es mayor o menor que nuestro valor ‘x’, una vez con esa definición, el algoritmo filtra los valores que no requiere y se mueve al lado que en el que sí encaja el valor ‘y’ y así sucesivamente. La razón por la que este método es tan efectivo es porque en cada paso, el proceso elimina la mitad del arreglo seleccionado mientras que la búsqueda lineal tiene que ir uno por uno. Además, la búsqueda binaria tiene una complejidad de $O(\log n)$ que, bajo comparación, es mejor que la complejidad $O(n)$ de la búsqueda lineal.

Siguiendo un pensamiento lógico, es posible asegurar que, como la búsqueda binaria elimina la mitad (50%) del arreglo con cada paso que da y la búsqueda lineal tiene que recorrer el arreglo completo (100%), la búsqueda binaria tarda al menos la mitad del tiempo que tardaría la búsqueda lineal, en especial si se trata de un arreglo, vector o lista larga como la que tenemos en nuestra bitácora original. Tomando las dos opciones en cuenta, no es difícil encontrar la solución más efectiva para nuestro reto.

d) Conclusiones

Parte de nuestro trabajo como programadores consiste en determinar cuál de todos los caminos es el más eficiente y fácil de comprender para nuestros clientes, tomando también en cuenta que este debe ser un método consistente que no falle cuando pasan diferentes cantidades de información. En esta primera parte del reto aplicamos esta teoría para filtrar y ordenar una bitácora con una cantidad grande de datos que tenían que presentarse al usuario, claro, sin tardar demasiado tiempo. Para esto tuvimos que evaluar nuestras opciones en los algoritmos a usar y compararlos en cuanto a resultados, tiempo y complejidad. A lo largo de esta actividad aprendí los muchos métodos que se pueden utilizar tanto para ordenar como para buscar información. Tuve muchas dudas, pero con asesorías, se pudo resolver el problema. Había tenido mucha dificultad con entender cómo acceder a la información de un archivo aparte a través del código para poder usar los datos

dentro de él desde el semestre pasado, pero ahora he aprendido mucho más y creo que es algo que ya puedo implementar sola.

Una de las áreas de oportunidad que me gustaría mencionar es que me gustaría estudiar más tiempo sola (ver videos, leer, hacer ejercicios extra) para llegar con dudas más concretas a clases y asesorías y poder implementar más código yo sola sabiendo lo que estoy haciendo. Programar es algo que disfruto mucho una vez que entiendo lo que debo hacer, espero que con las actividades que haremos a lo largo de esta materia pueda agarrar más la onda de cómo es que funcionan estos algoritmos.

e) Referencias

- GeeksforGeeks (2022) *Bubble Sort*. Recuperado el 29 de Mayo, 2022 de <https://www.geeksforgeeks.org/bubble-sort/?ref=lbp>
- GeeksforGeeks (2022) *QuickSort*. Recuperado el 29 de Mayo, 2022 de <https://www.geeksforgeeks.org/quick-sort/?ref=leftbar-rightbar>
- GeeksforGeeks (2022) *Linear Search*. Recuperado el 02 de Junio, 2022 de <https://www.geeksforgeeks.org/linear-search/>
- GeeksforGeeks (2022) *Binary Search*. Recuperado el 02 de Junio, 2022 de <https://www.geeksforgeeks.org/binary-search/>

Actividad 2.3 (Estructura de datos lineales)

a) Estructura del código

En esta actividad, hemos reemplazado lo que es nuestro vector `ListaRegistros` con una lista doblemente ligada bajo el mismo nombre. Para esto, se añadieron algunos archivos llamados:

- `DLLNode`: Una lista ligada, a diferencia del vector con el que estábamos trabajando en la actividad anterior, utiliza espacios llamados nodos para almacenar información. Es similar a una cadena, donde cada eslabón o *link*, como se diría en el inglés, es un espacio independiente que guarda un dato en específico. Para movernos entre nodos es necesario declarar los apuntadores que nos llevarán de eslabón en eslabón, y es para eso que existe nuestro archivo `DLLNode.h`. En este archivo tipo `.h` declaramos la clase `DLLNode` y dentro de ella declaramos 2 apuntadores del tipo `DLLNode` llamados *prev* (previous) y *next*, apuntadores que nos ayudarán a navegar entre los nodos de la lista ligada. También incluye la declaración de un nodo, como una función que nos servirá después para llamar para crear un nodo en nuestra lista.
- `DLinkedList`: El archivo `DLinkedList.h` es considerablemente más extenso que el archivo donde declaramos nuestros nodos. Esto es debido a que ahora toca desarrollar nuestra clase `DLinkedList` con sus funciones y respectivos apuntadores como lo son *head* y *tail* (inicio y fin de nuestra lista ligada) además de los nodos que nos permitirán desarrollar nuestras funciones como por ejemplo el nodo *partition* que es el nodo que representa la mediana de la lista ligada. Entre las funciones que están en este archivo, están:

<code>DLinkedList</code>	No cae realmente en la categoría de ser una función, sino que forma parte de la clase (constructor). Sin embargo, cumple la acción de crear una nueva lista vacía que declara los valores de los apuntadores <i>*head</i> y <i>*tail</i> como nulos y el número de elementos en la lista empieza siendo 0. Esta función no recibe ningún parámetro.
<code>~DLinkedList</code>	Esta función se encarga de “reiniciar” una lista que ya está declarada. Elimina los valores dentro de ella y restablece los valores de los nodos de inicio y fin como nulos junto con su número de elementos que regresa a ser 0. Esta función no recibe ningún parámetro.
<code>GetNumElements</code>	La función <i>GetNumElements</i> tiene sólo una línea de código que pide la devolución del valor entero de elementos que se encuentran en la lista ligada. Esta función no recibe ningún parámetro.
<code>printList</code>	Como lo dice su nombre, la tarea de esta función es imprimir los elementos de la lista en el mismo formato de lista ligada. El proceso va de nodo en nodo extrayendo e imprimiendo los valores que se

	encuentran en cada uno. Esta función no recibe ningún parámetro.
<code>printLastNode</code>	Esta función imprime sólo el valor que se encuentra en el nodo <i>*tail</i> que en una lista ordenada sería el valor más alto o más bajo de la lista. Al igual que las funciones anteriores, esta no requiere ningún parámetro para funcionar.
<code>addFirst</code>	Esta función nos permite añadir un nodo al principio de la lista. Para esto, se crea un nuevo nodo para el <i>*head</i> de la lista y se recorren los valores a un valor de índice mayor (i++) y se añade un valor a nuestra variable <i>numElementos</i> . Aquí sí requerimos el parámetro llamado "value" que permite introducir el valor que tiene el nodo nuevo.
<code>addLast</code>	Esta función nos permite añadir un nodo al final de la lista. Para esto, se crea un nuevo nodo para el <i>*tail</i> de la lista y se añade un valor a nuestra variable <i>numElementos</i> . Aquí sí requerimos el parámetro llamado "value" que permite introducir el valor que tiene el nodo nuevo. En el caso de que los nodos <i>*head</i> y <i>*tail</i> estén vacíos en un principio, se llama a la función <i>addFirst</i> para dar inicio a la lista.
<code>deleteData</code>	La función <i>deleteData</i> existe para eliminar cualquier dato dentro de nuestra lista y reacomodar los nodos de información alrededor del hueco. El dato a eliminar puede encontrarse en el <i>*head</i> , <i>*tail</i> o en medio de la lista. Esta función sí requiere el parámetro llamado "value" que es el valor dentro del nodo que deseamos eliminar de nuestra lista.
<code>deleteAt</code>	Esta función también sirve para eliminar datos pero, a diferencia de la función <i>deleteData</i> , este proceso elimina un nodo por su posición en la lista y no por el valor que tiene el mismo. Esta función requiere el parámetro llamado "position" que es el índice del nodo que deseamos eliminar de nuestra lista.
<code>getData</code>	La función <i>getData</i> nos regresa sólo el valor de un nodo en la lista en una posición o índice específico. Si la posición está fuera del rango de <i>numElementos</i> , se imprime un mensaje de aviso/error. Esta función requiere el parámetro llamado "position" que es el índice del nodo que deseamos ver de nuestra lista.
<i>*getHead</i>	La función <i>*getHead</i> tiene sólo una línea de código que pide la devolución del nodo que se encuentra en el primer lugar en la lista ligada. Esta función no recibe ningún parámetro.

partition	Es el primer paso para realizar el quicksort. El <i>partition</i> saca el valor medio del arreglo (o de la lista en este caso) y lo utiliza como punto pivote para el ordenamiento. Como parámetros, la función requiere del nodo con el valor más bajo del arreglo y el nodo con el valor más alto del mismo.
quickSort	Como se mencionó en la reflexión de la actividad anterior, el <i>quickSort</i> utiliza el método “divide y vencerás”, por lo que llama a la función <i>partition</i> y lo utiliza para poner los valores mayores de un lado de la mediana y los menores del otro y así consecutivamente hasta que el listado completo está ordenado. Igual que el <i>partition</i> , la función <i>quickSort</i> requiere de los nodos *low y *high para hacer su trabajo.
callQuickSort	Esta función se encarga meramente de acomodar la función de <i>quickSort</i> y prepararla para su uso. No requiere de ningún parámetro porque todos los que necesita ya se llamaron en la función <i>quickSort</i> .
middle	La función <i>middle</i> se ocupa de encontrar la mediana de un arreglo ordenado para poder utilizar ese valor como punto pivote en la función <i>binarySearch</i> .
*binarySearch	Finalmente, se adaptó la función <i>binarySearch</i> para que pudiera usarse con la lista ligada que implementamos para esta actividad. Utiliza la función <i>middle</i> para dividir el arreglo en dos y determinar si el valor que se busca es mayor o menor que el valor middle y se dirige a un lado de la lista dependiendo de eso.

b) Listas enlazadas

El cambio principal que se realizó en esta actividad fue reemplazar el vector que se estaba utilizando para almacenar los datos de la bitácora con una lista enlazada que funciona a base de nodos y establecer los métodos adecuados para su manejo en el programa. Sin embargo, esto trae la pregunta, ¿por qué el reemplazo del arreglo (vector) con una lista enlazada?

Uno de los beneficios principales de una lista enlazada es la flexibilidad con la que cuenta. Un arreglo tiene una capacidad fija desde que se declara y, aunque es posible añadir elementos adicionales a un arreglo, es un proceso largo de acomodar, espaciar y finalmente añadir un valor con la esperanza de que se haya añadido de manera exitosa. No es el método más eficiente ya que, además, existe el riesgo de perder alguno de los datos originales del arreglo en el caso de que se sobreescriba algún valor nuevo encima de otro y tendremos un desastre que acomodar. Mientras que los arreglos tienen ese problema, las listas enlazadas se encuentran muy tranquilas del otro lado del espectro. Una lista enlazada está dispuesta a expandirse

en cualquier momento dependiendo de la cantidad de datos que se ingresen al programa. Al trabajar con nodos, establecer funciones para añadir un nodo más o incluso borrar cualquiera de los nodos ya existentes es muy sencillo. En vez de estar trabajando con un índice fijo que acomoda todos los datos de manera consecutiva, el acomodo de los datos en una lista enlazada es mucho más libre, cuidando, por supuesto, que los nodos no se sobreescriban con los métodos de dirección abierta (sea por prueba lineal o prueba cuadrática) para encontrar el mejor lugar para el dato en cuestión. En adición a la eficiencia de añadir o eliminar valores de una lista enlazada, esta misma flexibilidad que se demuestra en las listas aumenta la eficiencia de uso de memoria. Esto es debido a que en vez de tener el arreglo completo, el programa va leyendo la lista nodo por nodo y estos ocupan menos espacio en la memoria que un arreglo.

En esta actividad se hizo uso de una lista doblemente enlazada. ¿Cuál es la diferencia entre una lista enlazada y una lista doblemente enlazada? ¿Por qué es que la doblemente enlazada fue más efectiva para este trabajo? Una lista doblemente enlazada nos permite tener dos apuntadores para movernos a lo largo de la lista, estos son **prev* y **next*. Estos apuntadores nos permiten movernos en la dirección que gustemos en la lista, hacia adelante de la línea o hacia atrás, mientras que una lista enlazada simple sólo nos deja movernos hacia adelante y, si necesitamos un dato que está atrás, debemos dar la vuelta a todos los nodos para regresar al inicio y de ahí encontrar el nodo que buscamos. Ambos tipos de listas enlazadas son muy efectivas, pero cuando estamos trabajando con un archivo tan extenso como el que tenemos ahora (el archivo de bitácora) tener que estar regresando al inicio cada que queramos un cierto elemento tomará demasiado tiempo y memoria de nuestro programa.

Haciendo una prueba entre las actividades 1.3 (arreglos) y la actividad 2.3 (listas enlazadas) se descubrió una diferencia de tiempos significativa entre las 2. El programa de la actividad 1.3 tardó 16.27 segundos, mientras que el programa de la actividad 2.3 tardó 10.12. Cuando lo vemos así, parece que la diferencia no es tan grande entre los tiempos de ambas actividades, pero cuando se trata de proveer la mayor eficiencia para un cliente, 6 segundos pueden ser muy valiosos para la persona utilizando el sistema.

A través de esta actividad, podemos observar los múltiples beneficios que tienen las listas enlazadas. La flexibilidad que proporcionan pueden ser un apoyo si la lista de la bitácora va creciendo todos los días mientras que si tenemos un arreglo, tendríamos muchos problemas si la lista de IPs se actualiza seguido.

c) Conclusiones

El mundo de la programación y tecnología siempre se está actualizando. Siempre hay innovaciones y cambios para tener todo un menú de opciones para componer nuestros sistemas. En esa gran gama de opciones se incluyen lo que son las estructuras de datos, que es la categoría donde entran los arreglos, listas enlazadas (simples y dobles), BSTs (que veremos después), etc. Tenemos todas estas posibilidades para armar nuestro código. Es como armar nuestro propio Lego. Le podemos dar la forma que queramos, del mundo que queramos, pero debe tener un propósito y, si no lo tiene, es muy probable que una de las piezas que acomodemos

esté chueca y se tenga que reparar si no queremos arriesgar que se caiga toda la estructura. En el caso de esta actividad, se realizaron modificaciones para introducir las listas ligadas para introducir con ellas todos los beneficios mencionados más arriba. Claro, todo tiene sus pros y contras. Por ejemplo, en el caso de requerir un elemento en específico de nuestra estructura, es mucho más fácil llamarlo de un arreglo por su índice que ir nodo por nodo buscando el dato para regresarlo. De cualquier manera, las listas ligadas son un aliado formidable que nos apoyará a futuro cuando los trabajos o programas que se nos sean requeridos para la escuela o ya para el mundo laboral tengan los archivos enormes que tengamos que filtrar o acomodar dentro del sistema.

Todo tiene su uso, los arreglos no han sido descalificados en su uso, sino simplemente estamos aprendiendo a evaluar para qué podemos utilizar cada estructura de datos para que, en el futuro, tengamos esa competencia laboral y podamos ser lo más efectivos posibles con nuestro tiempo y nuestra programación.

d) Referencias

- GeeksforGeeks (2022) *Linked List vs. Array*. Recuperado el 06 de Junio, 2022 de <https://www.geeksforgeeks.org/bubble-sort/?ref=lbp>

Actividad 3.4 (Binary Search Trees)

a) Estructura del código

Esta actividad consistió en establecer otra estructura también conocida como un Árbol Binario. Como dice su nombre, la estructura depende de nodos y de cada uno hay un máximo de 2 nodos salientes. El ordenamiento de los nodos depende del heap que se haya elegido para el sistema en particular. Un heap es una estructura de datos que se basa en los árboles y se puede elegir sea un MaxHeap (que el valor más alto esté en la posición del tronco) o un MinHeap (que el valor menor sea el tronco). Para esta actividad se seleccionó el uso de un MaxHeap con el propósito de que en la actividad se nos pide imprimir los 5 valores más altos de los accesos y es mucho más fácil acceder al tronco que a las ramas. Para lograr el funcionamiento de esta actividad se añadieron algunos archivos y funciones más.

- *mhPair*: En los archivos *.h* y *.cpp* *mhPair* introducimos un nuevo objeto llamado igual que el mismo archivo. El objeto *mhPair* es un par del MaxHeap, esto quiere decir que almacena 2 celdas de información y nos permite manipularlas como cualquier otro tipo de dato. Para esta actividad, nuestro par guarda la información de un número entero llamado *accesos* (es un contador de la cantidad de accesos que ha tenido esa IP) y la misma *ip* pero ahora en string. Además de declarar el nuevo objeto *mhPair*, estos archivos declaran también la sobrecarga de operadores para que el manejo de nuestro *mhPair* sea más sencillo. En cuanto a funciones, dentro del *.cpp* se implementan los métodos *getAccesos()* y *getIP()*. Estas funciones tienen una acción muy sencilla, cuando se les llama, estos regresan el valor que tiene la variable correspondiente (*accesos* o *ip*).
- MaxHeap:

MaxHeap() MaxHeap(int capacity)	No cae realmente en la categoría de ser una función, sino que forma parte de la clase (constructor). Sin embargo, cumple la acción de crear un nuevo heap vacío que declara los valores <i>size</i> y <i>maxSize</i> como nulos (0). Esta función puede recibir un parámetro o podemos esperar a declarar la capacidad máxima del heap en cuestión.
~MaxHeap()	El destructor de nuestra clase se encarga de “reiniciar” un heap que ya está declarada. Elimina los valores dentro de ella y restablece los valores de las celdas como nulos junto con su número de elementos que regresa a ser 0. Esta función no recibe ningún parámetro.
isEmpty()	En el caso de que un heap esté vacío, entra a jugar la función <i>isEmpty</i> . Compuesta de sólo una línea de código, esta declara que si el valor de <i>size</i> en nuestro programa es igual o menor a 0, regresa el resultado al usuario para avisar que el heap con el cuál se está trabajando está vacío.

<code>getCapacity()</code>	La función <i>getCapacity</i> tiene sólo una línea de código que pide la devolución del valor entero de elementos que se encuentran en la lista ligada. Esta función no recibe ningún parámetro.
<code>setCapacity(int capacity)</code>	En esta actividad estamos trabajando con arreglos nuevamente (por eso es que más abajo podemos trabajar con índices). En la reflexión anterior se mencionó que en los arreglos añadir y eliminar información puede ser tedioso ya que tenemos un valor de capacidad definido desde el principio, pero con la función <i>setCapacity</i> tenemos un poco más de tiempo para definir el espacio máximo que tendrá nuestro heap. Esta función requiere de un valor entero <i>capacity</i> que será ingresado para determinar el valor de la capacidad del heap.
<code>printMaxHeap()</code>	Como dice su nombre, la función <i>printMaxHeap</i> permite imprimir el árbol con todo y valores. Este método cuenta solamente con un for muy sencillo que establece que mientras el el valor [i] esté por debajo del valor del tamaño del MaxHeap, imprima el elemento que se encuentra en el índice [i]. Esta función no recibe ningún parámetro.
<code>push(T key)</code>	Esta función nos permite añadir un nodo al final de nuestro heap. Para esto, se evalúa si el heap ya está lleno o si aún hay espacio para el nuevo valor y añade un valor a la variable <i>size</i> . Para este proceso, se requiere del parámetro <i>key</i> que representa el valor que se va a ingresar. Después de incluir el valor en el listado, se verifica que ninguna de las reglas del heap hayan sido violadas.
<code>top()</code>	Nuestro <i>top</i> es una de las funciones más importantes de este programa, ya que nos permitirá encontrar el valor más alto de nuestro heap que, para sacar el par que más accesos haya tenido, lo necesitamos al menos para la primera parte. Su trabajo es muy sencillo, ya que sólo evalúa si el heap está vacío (en el caso de que no, envía un mensaje de que se encuentra fuera de rango) y si tiene valores, regresa el primer dato del MaxHeap, que también es el valor más alto de nuestra estructura. Esta función no recibe ningún parámetro.
<code>heapifyDown()</code>	La función <i>heapifyDown</i> se utiliza para cuando se elimina el valor más alto de nuestro heap, por lo que va de la mano con la función <i>pop</i> . Tras eliminar el valor que representa nuestro tronco, los datos se deben reacomodar para regresar a un estado balanceado. Para esto, evalúa los valores que se encuentran alrededor, selecciona el nuevo valor

	más alto y lo pone al principio. De ahí, si los valores son menores al nodo que se tiene seleccionado, irán a la izquierda y si son mayores (que el número de la izquierda pero no mayores que nuestro <i>top</i>) se colocan del lado derecho. Esta función no recibe ningún parámetro.
<code>pop()</code>	En el caso de que un MaxHeap tenga datos, la función <i>pop</i> toma el valor más alto del MaxHeap (el <i>top</i>) y lo elimina para dar lugar a que un nuevo valor tome su lugar en la estructura. Como sabemos que el valor más alto se encuentra en la primera posición del MaxHeap, por lo que eliminamos el contenido de esa casilla y pasamos el valor de la última casilla a la primera posición. Para no dejar nada desacomodado, se elimina la última celda y se vuelve a hacer <i>heapifyDown()</i> del árbol para que todo quede balanceado nuevamente. Esta función no recibe ningún parámetro.
<code>getSize()</code>	La función <i>getSize</i> tiene sólo una línea de código que pide la devolución del valor entero de elementos que se encuentran en la lista ligada. Esta función no recibe ningún parámetro.

- Bitácora: Esta sección ya se había establecido desde el principio, pero con las nuevas estructuras, ha habido algunos cambios. Principalmente, se añadieron algunas funciones más para apoyar con el manejo del heap. Estas son:

<code>heapify(int n, int i)</code>	Similar al <i>heapify</i> anterior, el deber de la función es transformar los datos en un heap (estructura de árbol). En este caso, que estamos usando MaxHeap, el valor más alto de todo el listado ocupa la primera posición y de ahí la función va evaluando de qué lado debe ir cada valor para que todo quede acomodado y balanceado. Este <i>heapify</i> recibe los parámetros enteros <i>i</i> y <i>n</i> . Estos valores serán puntos de apoyo para que el heap quede arreglado de manera correcta
<code>heapSort()</code>	Como dice su nombre, el <i>heapSort</i> se encarga de ordenar los datos del heap. Nuestro <i>heapify</i> puede lograr acomodar los valores hasta cierto punto y, para que no tengamos problemas después, nuestra función <i>heapSort</i> reevalúa si el heap está ordenado correctamente o si hay que realizar modificaciones. Esta función no recibe ningún parámetro.
<code>cuentaRepetidos(std::string filename)</code>	La función que le dará sentido y nos permitirá cumplir con lo que se nos pidió en esta actividad

	<p>está al último. Ya que tenemos nuestro heap ordenado por IPs, podemos ir contando cuántas veces se repite una cierta dirección IP y así, lograr tener el orden de las que más se repiten a las que menos. Tras esto, podemos hacer un <i>top</i> (para imprimir el que tiene más accesos) y luego un <i>pop</i> para imprimir el siguiente y así tener nuestro TOP 5 de IPs con más accesos.</p>
--	---

b) Heap y MaxHeap

Tomando en cuenta que en la actividad pasada trabajamos con listas enlazadas y ahora nos movimos a heap, ¿por qué el cambio si las listas enlazadas eran lo máximo? Como se mencionó en la reflexión anterior, tenemos muchas opciones cuando se trata de estructuras de datos para nuestros programas y cada una es buena para algo en específico. En esta actividad se nos pidió ordenar nuestra bitácora de acuerdo a las direcciones IP y luego contar cuántas veces se registra cada una para después imprimir las 5 direcciones IP que más se repiten en la bitácora. El MaxHeap fue la mejor opción aquí ya que, en el mismo ordenamiento, nos sale el valor más alto en el lugar más accesible, el principio. De esta manera, pudimos implementar un for para que el proceso de imprimir el valor más alto y luego eliminarlo para dejar pasar el siguiente valor se repita 5 veces. El heap tiene las funciones que nos permiten trazar este proceso (*top* y *pop*) y a partir de ahí podemos continuar con nuestro trabajo.

La creación de un heap tiene una complejidad de $O(n)$, que es bastante decente y dentro del mismo heap se pueden utilizar las priority queues, que son funciones sencillas para usar con un heap con complejidad de tiempo de $O(\log n)$ para manejarlo. Es un pequeño truco que nos servirá después. Un heap también tiene un ordenamiento natural. Aunque tenemos que también implementar el *heapSort* para ordenarlo cuando se mueva algo, la estructura a partir del *heapify* ya tiene un orden, sea de MaxHeap o de MinHeap.

Otra de las cosas que fue de gran utilidad en esta actividad fue que realmente aprovechamos el uso de los arreglos. Las listas ligadas fueron una excelente solución para la actividad pasada, sin embargo, para esta actividad hubiera sido muy cansado y poco efectivo tener que pasar la lista una y otra vez en búsqueda de un elemento de la lista. En nuestra bitácora de grafos que usamos en este trabajo tenemos 105,281 registros y no sabemos cuántos están repetidos y, aunque lo estén, no se reducirá el número por mucho. Con una lista a fuerzas tenemos que empezar a recorrerla desde el **head*, no se puede empezar desde la mitad de la lista, entonces se pierde mucho tiempo en lo que la búsqueda inicia y reinicia para buscar otro dato. Pero, con los arreglos que empleamos con el heap, pudimos usar los índices para encontrar los datos con mayor facilidad y se necesitó menos tiempo para que el programa nos pasara la información que requerimos.

c) ¿Cómo saber si una red está infectada o no?

El primer paso que tomamos para descubrir cuáles direcciones tenían mayor probabilidad de sufrir algún acceso maligno fue imprimir de nuestra bitácora las 5 direcciones IP con más accesos. Por lógica, es posible que mientras más intentes acceder a algo, más probabilidades tendrás de acceder en cierto punto. Como un ejemplo algo cómico, mis primos van a mi casa todos los viernes y tratan de adivinar la contraseña de mi celular. Cuando hablamos de lógica, es mucho más probable que accedan a mi celular si cada viernes hay un intento de entrar a que si no pudieran agarrar mi teléfono y, por lo tanto, no poder intentar nada. Es lo mismo con estas IPs. Hay mucha más probabilidad de que se logre acceder a un dispositivo si se intenta acceder 18 veces a si sólo se accedió 1 vez. Es con esta mentalidad que se desarrolló el programa que imprimiera las 5 IPs con más accesos para descubrir las 5 IPs con mayor probabilidad de ser infectadas.

Algunos métodos simples para saber si nuestro dispositivo, cualquiera que sea (no tiene que ser una computadora), está infectado se enfoca principalmente en la lentitud del internet o sistema en general. Nuestro dispositivo tardará más de lo usual en conectarse al internet, en abrir aplicaciones, prenderse o apagarse. Además, una de las luces rojas de las cuales nos damos cuenta es cuando empezamos a recibir mails o mensajes de spam. En el Tec ocurre mucho que llegan mails de lo que parecen ser personas que conocemos (compañeros, maestros, staff) con un archivo que nos invita a abrirlo, pero contiene un virus y tenemos muchos problemas con la computadora después. También se puede notar cuando nuestros contactos empiezan a recibir mensajes con links, imágenes, archivos, etc. que no enviamos nosotros. Esto suele ocurrir mucho a través de Facebook y Messenger. Es por esto que debemos tener mucho cuidado de las páginas que abrimos y los archivos que descargamos para no tener este caso.

d) Conclusiones

El tema de ciberseguridad está haciendo un boom en el mundo de la tecnología.

Conforme van aumentando las innovaciones en las diferentes tecnologías, van creciendo también las amenazas que se esconden en la web. Ahorita tenemos este problema de que se nos pueden enviar correos amenazantes incluso desde la red de la institución, que debe ser muy segura. Los cibercriminales se aprovechan del surgimiento de nuevos métodos y lenguajes que les permiten estar más cerca de nuestros dispositivos e información personal que pueden vender después.

De experiencia propia, se me ha avisado que las contraseñas en mi celular han sido vulneradas y después me han empezado a marcar intentando extorsionar o mensajes que tratan de invitarme a x o y empresa. Ya no tuve de otra más que estar bloqueando y bloqueando números que me estaban marcando.

Dentro de nuestra carrera está la concentración de ciberseguridad que incluye los estudios en lo que es el hacking ético. Ahora mismo, esa es un área que muchas empresas están buscando, ya que es necesario que tengan a alguien que cuide la información confidencial de la misma compañía y de sus empleados ya que tienen la obligación de proteger su información personal. Actualmente es una de las áreas que más me gustan de la carrera y posiblemente la que me gustaría estudiar cuando lleguemos al semestre Tec.

e) Referencias

- Manwani, C. (2022) *Time complexity of building a Heap*. Recuperado el 13 de Junio, 2022 de <https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/>
- GeeksforGeeks (2018) *Applications of Heap Data Structure*. Recuperado el 13 de Junio, 2022 de <https://www.geeksforgeeks.org/applications-of-heap-data-structure/>
- Jiménez, J. (2020) *Cómo detectar si nuestro equipo es una Botnet y qué hacer*. Recuperado el 13 de Junio, 2022 de <https://www.redeszone.net/tutoriales/seguridad/detectar-equipo-botnet-seguridad/>
- Lorenzo, J.A. (2021) *Cómo saber si mi IP forma parte de una botnet y cómo evitarlo*. Recuperado el 13 de Junio, 2022 de <https://www.redeszone.net/tutoriales/seguridad/comprobar-ip-publica-esta-botnet/>

Actividad 4.3 (Grafos)

a) Estructura del código

- `ipAddress`: como lo hicimos en la actividad 3.4, creamos un objeto llamado `ipAddress` que podremos manipular con las funciones dentro de los archivos bajo el mismo nombre (`.h` y `.cpp`). Este objeto nos apoyará cuando se trate de imprimir una cierta información de nuestros nodos. Los archivos en los cuales están desarrollados los métodos incluyen, además de la sobrecarga de operadores que nos permitirán la comparación de elementos, las siguientes funciones:

<code>getIp()</code>	La función <i>getIp</i> tiene sólo una línea de código que pide la devolución del valor string de la dirección IP donde se encuentra el apuntador. Esta función no recibe ningún parámetro.
<code>getIpPort()</code>	La función <i>getIpPort</i> tiene sólo una línea de código que pide la devolución del valor string del puerto de la dirección IP donde se encuentra el apuntador. Esta función no recibe ningún parámetro.
<code>getIpValue()</code>	La función <i>getIpValue</i> tiene sólo una línea de código que pide la devolución del valor int (numérico) de la dirección IP donde se encuentra el apuntador. Esta función no recibe ningún parámetro.
<code>getIpIndex()</code>	La función <i>getIpIndex</i> tiene sólo una línea de código que pide la devolución del valor int (numérico) de la posición de la dirección IP donde se encuentra el apuntador. Esta función no recibe ningún parámetro.
<code>getDegree()</code>	La función <i>getDegree</i> tiene sólo una línea de código que pide la devolución del valor int (numérico) del grado (vértices salientes del nodo donde nos encontramos). Esta función no recibe ningún parámetro.

- `Graph`: ahora estamos introduciendo el tema de grafos, cuyos métodos nos permiten acceder a un nodo por diferentes vías. En nuestro programa, tenemos un `.h` y un `.cpp` para los grafos donde declaramos las funciones y variables que utilizaremos para esta actividad. Las principales de estas son:

<code>saveDegrees(std::string filename)</code>	La función <i>saveDegrees</i> se encarga de almacenar los pares de direcciones IP y el grado (función <i>getDegree</i>). Esta función requiere del parámetro <code>filename</code> para que el documento de salida (resultado) tenga un nombre.
--	--

<code>saveTop(std::string filename)</code>	La función <i>saveTop</i> se asimila a la función <i>saveDegrees</i> en que también guarda el resultado en un archivo aparte, pero esta función guarda como resultado las 5 IPs con mayor cantidad de accesos (las IPs con mayor riesgo de malware) y además imprime en pantalla la IP del Bootmaster, o la IP con mayor cantidad de accesos. Esta función requiere del parámetro <code>filename</code> para que el documento de salida (resultado) tenga un nombre.
<code>shortestPath(std::string filename)</code>	Finalmente, la función <i>shortestPath</i> cuenta los saltos entre nodos para averiguar a qué distancia está cada nodo de la IP del Bootmaster. Esto es para ver cuáles son las IPs en mayor riesgo y cuál es la que está en menor riesgo. Esta función requiere del parámetro <code>filename</code> para que el documento de salida (resultado) tenga un nombre.

b) Grafos

Los grafos son una estructura de datos no-lineal, lo que quiere decir que no sigue una sola línea de pensamiento o de recorrido. En el caso de los grafos, cada nodo puede tener múltiples salidas y múltiples entradas a y de otros nodos, de manera que se hace una enorme bola de estambre de nodos interconectados.

Parte de nuestros métodos es el algoritmo de *shortestPath* de Dijkstra que es un procedimiento de complejidad media. La complejidad de tiempo de este método es de $O(V)$, V siendo la cantidad de vértices que hay en el grafo. Hace sentido que mientras más grande sea el grafo, más mezclado va a estar y más difícil será desenredarlo. Esto se ha tomado en consideración ya que parece ser el método más completo y directo para solucionar el problema. Es como un problema de Pinball. Va rebotando de nodo a nodo contando cuántos saltos hay que dar para llegar de un nodo base (que es nuestra IP del Bootmaster) hasta todos los demás nodos.

Ahora, para explicar lo que es la IP del Bootmaster. Por 2 actividades ya hemos estado sacando lo que son las TOP 5 IPs más accesadas y que, por ello, son las IPs que se encuentran en mayor riesgo de ser pirateadas. Pero la que ocupa el primer lugar, la que está en mayor peligro se llama la IP del Bootmaster. Para esta actividad se pide que imprimamos en pantalla esta IP junto con el nodo en el que se encuentra (para esto usamos la función *getIpIndex*). La IP del Bootmaster será la que más entrantes tenga desde diferentes IPs, básicamente es cuando todas parecen estar apuntando a esa misma IP.

```
➤ make -s
➤ ./main
73.89.221.25 con grado 18
185.109.34.183 con grado 18
90.167.116.74 con grado 17
64.154.86.234 con grado 17
220.210.239.189 con grado 17

El bootmaster es: 73.89.221.25
Se encuentra en el nodo: 3916

Distancia máxima: 922
IP a distancia máxima: 244.223.133.50
➤ █
```

c) Conclusiones

Los grafos pueden ser un poco difíciles de comprender por el enredadero que viene con ello, sin embargo, una vez que lo vemos a detalle, pueden ser muy divertidos y satisfactorios de manejar. Estos nos permiten asegurar que nuestra información esté segura y bien acomodada. Los múltiples usos de los grafos nos apoyan en el movimiento y trabajo de datos que, en este caso, se representan por los rebotes entre direcciones IP de la bitácora que nos muestran cómo se recorren estas direcciones cuando se trata de hacking o cuando sentimos que nuestras IPs están bajo peligro.

De esta actividad me llevo un ánimo presente para estudiar más a profundidad sobre los temas de esta clase y trabajar en comprender por completo este tema.

d) Referencias

- Pandey, M. (2021) *Dijkstra Algorithm*. Recuperado el 17 de junio del 2022 de <https://www.scaler.com/topics/data-structures/dijkstra-algorithm/#code-482-4-0--time-complexity>