INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

TC1031

Reflexión

“Actividad 2.3”

Alumna:

Sarah G. Martínez Navarro

A01703113

Fecha:

06 / Junio / 2022

Profesor:

Eduardo Arturo Rodríguez Tello

ÍNDICE

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 archivo 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:

DLinkedList	No cae realmente en la categoría de ser una función, sino que forma parte de la clase que tiene el mismo nombre. Sin embargo, cumple la acción de crear una nueva lista vacía que declara los valores de los apuntadores **head* y **tail* como nulos y el número de elementos en la lista empieza siendo 0. Esta función no recibe ningún parámetro.
~DLinkedList	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.
GetNumElements	La función *GetNumElements* 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.
printList	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.
printLastNode	Esta función imprime sólo el valor que se encuentra en el nodo **tail* 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.
addFirst	Esta función nos permite añadir un nodo al principio de la lista. Para esto, se crea un nuevo nodo para el **head* de la lista y se recorren los valores a un valor de índice mayor (i++) y se añade un valor a nuestra variable *numElementos*. Aquí sí requerimos el parámetro llamado “value” que permite introducir el valor que tiene el nodo nuevo.
addLast	Esta función nos permite añadir un nodo al final de la lista. Para esto, se crea un nuevo nodo para el **tail* de la lista y se añade un valor a nuestra variable *numElementos*. Aquí sí requerimos el parámetro llamado “value” que permite introducir el valor que tiene el nodo nuevo. En el caso de que los nodos **head* y **tail* estén vacíos en un principio, se llama a la función *addFirst* para dar inicio a la lista.
deleteData	La función *deleteData* 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 **head*, **tail* 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.
deleteAt	Esta función también sirve para eliminar datos pero, a diferencia de la función *deleteData*, 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.
getData	La función *getData* 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 *numElementos*, 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.
*getHead	La función **getHead* 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 *partition* 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 *quickSort* utiliza el método “divide y vencerás”, por lo que llama a la función *partition* 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 partition, la función *quickSort* requiere de los nodos *low y *high para hacer su trabajo.
callQuickSort	Esta función se encarga meramente de acomodar la función de *quickSort* y prepararla para su uso. No requiere de ningún parámetro porque todos los que necesita ya se llamaron en la función *quickSort*.
middle	La función *middle* se ocupa de encontrar la mediana de un arreglo ordenado para poder utilizar ese valor como punto pivote en la función *binarySearch*.
*binarySearch	Finalmente, se adaptó la función *binarySearch* para que pudiera usarse con la lista ligada que implementamos para esta actividad. Utiliza la función *middle* 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. Un 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