Adrián González Saldívar A00827845
01/12/2020

Reflexión Final

Act 1.3 - Actividad Integral de Conceptos Básicos y Algoritmos Fundamentales

Cada algoritmo de ordenamiento y búsqueda cuenta con una complejidad y eficiencia que depende de un factor diferente al otro. En cada algoritmo de ordenamiento y búsqueda se presentan diferentes niveles de complejidad: el mejor caso, el peor caso y en promedio. Los algoritmos comparten el resultado del nivel de complejidad del mejor caso, en este nivel solo se hace una comparación, esto quiere decir que en todos los algoritmos de búsqueda y ordenamiento en el mejor caso solo ocurre una comparación. La velocidad de ejecución, que es la eficiencia, depende del tamaño del arreglo, sin embargo este tamaño puede cambiar en relación del algoritmo, puede ser lineal, cuadrático, logarítmico. En esta actividad se utilizaron varios algoritmos de ordenamiento y búsqueda, algunos de estos fueron vistos en clase como el quicksort con su función conjunta y la búsqueda binaria. El quicksort tiene complejidad en el mejor caso logarítmica de n (O(nlogn)) y en su peor caso, el cual es el que se usa para representar la función, O(n^2) , al utilizar esta función se requiere una función adicional, su complejidad es lineal O(n), por otro lado la función de búsqueda binaria tiene una complejidad logarítmica O(log n). Al usar estos algoritmos de ordenamiento y búsqueda nos ahorramos tiempo al momento de correr el programa, esto se debe a la eficiencia (velocidad) que se está aplicando al arreglo.

El algoritmo de quicksort fue utilizado para ordenar los datos que se encontraban en el archivo “bitácora.txt.” y la función búsqueda binaria se utilizó para encontrar los datos que pedimos, como el mes inicial, día inicial, mes final y dia final, que son los rangos de búsqueda, un punto importante que me di cuenta en esta actividad y en la pasada es que para que la búsqueda binaria funcione correctamente ocupa trabajar con un arreglo previamente ordenado.Otras dos funciones fueron usadas, ambas tienen una complejidad (O(1)) y cumplen un trabajo similar, ambos trabajan afectando los valores de la fecha.

En esta situación problema nos encontramos con varios pequeños problemas, uno de ellos se centra en el uso del “struct” pero esto fue solamente al inicio cuando empezamos a investigar el uso, y las diferentes maneras de establecer sus variables, ya sea después de la llave “}” y antes del punto y coma “;” o declararlo en el main. Al principio no entendía cómo usarlo hasta que empecé a verlo como una clase o un grupo de elementos bajo el mismo nombre. Otro problema que en particular me enfrente yo fue más la implementación de leer el archivo que nada, esto se debe porque aunque me acordaba/sabía de varias líneas de código que son clave para el algoritmo, algunas las terminaba cambiando ya sea por declaración o el orden en donde los colocaba. Un problema que teníamos en esta parte fue al desplegar en la consola los datos buscados ya que en algunos casos salía un dato que tenía otra fecha fuera del rango del que se pedía, esto se arreglo cambiando el límite final del ciclo for. Otro problema que tuve su entender en su totalidad como funciona la complejidad del Quicksort, no fue hasta que mi compañero de equipo me explico la razón por el cual se establece como cuadrática, este es porque dentro de la función se llama a sí misma dos veces, aunque con una diferencia de parámetros, cuenta como una repetición de llamada, lo cual lo hace cuadrática.

Act 2.3 - Actividad Integral estructura de datos lineales

Una de las ventajas de una lista doblemente enlazadas es la habilidad que ofrece de acceder al valor previo y siguiente de un nodo. Implementando el uso de punteros se es posible representar los datos ligados de un inicio a fin desde que estos son entrado, sin embargo a comparación con la lista doblemente enlazada circular, la que fue utilizada en esta práctica no liga los datos del inicio con los finales, sino en ambos extremos apuntan a direcciones nulas indicando que no hay más datos. Otra ventaja de una lista de este tipo es que se puede acceder a un elemento desde ambas direcciones.

Considerando problemas con naturaleza similar a la situación planteada, lo que se quiere obtener es la creación de nodos, y ordenar estos de manera pedida en la cual se pueda ingresar los elementos en un método de búsqueda, para que ciertos datos que cumplen un requisito puedan ser extraídos, en este caso sería el ip buscado por el usuario.Una desventaja de esta estructura de datos en consideración

a este problema planteado es una más compleja a comparación con otras que se pueden resolver de forma simple con la implementación de vectores, lista encadenada, esta en comparación con la doble, hace el ordenamiento y la extracción de datos con una mayor eficiencia al utilizar un archivo grande de datos.

La mayoría de las complejidades presentes en el algoritmo son lineales O(n) y constantes O(1), dependiendo de la clase ya que en la clase Node la mayoría, sino todos los métodos son constantes. El algoritmo con mayor complejidad fue el bubble sort con cuadrada O(n^2). Como se puede notar la mayoría del algoritmo no llega a más de ser lineal lo cual es bueno ya que esto demuestra la eficiencia que tendrá las operaciones y esto reducirá el tiempo de procesamiento para obtener un resultado, por lo tanto siempre es bueno hacer observaciones para poder mejorar la eficiencia en cada método posible.

Act 3.4 - Actividad Integral de BST

Un ABB (árbol de búsqueda binaria) tiene varias ventajas en su implementación, una de ellas es su eficacia en la memoria, ya que esta no utiliza más memoria de lo que es necesaria. En un ABB todos los elementos del subárbol del lado izquierdo son menores que los elementos almacenados en el subárbol derecho son mayores. Es posible realizar esto ya que al insertar un nuevo dato este se compara con el dato almacenado en la raíz y/o padre, dependiendo de su nivel, el cual dirigirá su dirección a un subárbol izquierdo o derecho. Esto hace demostración de su habilidad en organizar los datos en una manera que permite ser accesible fácilmente. Otro punto que se debe de mencionar es que en este ABB sus elementos solo son almacenables una vez, por lo tanto no hay repetición de datos. Todo esto permite que la búsqueda de un dato, de un nodo, sea más eficiente, tome menos tiempo.

En esta situación problema fue una ventaja el utilizar un ABB, ya que al acomodar todos los nodos en el orden del que nos era pedido, se nos simplifica al acceder a los datos que se nos pedían imprimir, estos eran los más repetidos que por ende estarían en el subárbol derecho. Igualmente algo importante en el código fue el uso de la librería de queue y struct, el cual el segundo nos permitio guardar dos datos

relacionados de diferentes tipos en un mismo nodos y el primero nos ayudó a guardar y sacar los nodos buscados.

Las complejidades de los métodos de un ABB (BST) se basan en la altura del árbol, es decir, se basan en la cantidad de datos ingresados y la cantidad de niveles que contienen los subárboles izquierdo y derecho.

Act 4.3 - Actividad Integral de Grafos

Como se vio en las clases anteriores un grafo es un conjunto de nodos y de arcos, ambos igualmente son llamados como vértices y aristas respectivamente.Cuando se implementa un grafo se tiene que conocer justamente cuál es el tipo de grafo a utilizar en la situación adecuada, daré dos ejemplos de diferentes grados estos son el grafo dirigido y el no dirigido. Debido a la naturaleza de la situación problema que se nos fue pedida resolver se vio adecuado, la implementación de un grafo dirigido.

En un grafo dirigido los arcos tienen una dirección, a diferencia de un no dirigido que es bidireccional. Un grafo se puede implementar en base de una matriz de adyacencia o de una lista de adyacencia, esta última fue la utilizada en esta situación problema, cada una cuenta con sus ventajas y desventajas. Una de las ventajas del uso de la lista de adyacentes es que hace un buen uso de la memoria y es muy eficiente. Este se utiliza cuando la cantidad de arcos es menor a O(n^2). Su desventaja es en la eficiencia en determinar la existencia de un arco de un vértice a otro, al igual que requiere un mayor espacio de memoria. Este factor fue tomado en cuenta durante el desarrollo de la situación problema, es por eso que se agregó y utilizo la libreria de lo que se conoce como unordered_map, el cual tiene una eficiencia constante (O(1)), lo que significa que hace el proceso mucho más rápido por su implementación.

El algoritmo con mayor complejidad en esta situación problema fue de O(V + E). Lo que esto representa es que la complejidad del algoritmo está basada en la cantidad de vértices y aristas introducidas al programa. La menor complejidad en el algoritmo como antes mencionado es el O(1).

Act 5.2 - Actividad Integral sobre el uso de códigos hash

En actividades pasadas se utilizó un vector para demostrar un método de organización de datos provenientes de una tabla hash. En esta ocasión con esta situación problema optamos por implementar un “unordered_map” el cual contiene dos elementos, un string y un par de un entero y un vector de strings. Este unordered_map fue modificado a partir de su implementación original de un string, y un par de enteros. Como información complementaria cabe recalcar que el unordered_map tiene una complejidad nivel constante: O(1).

La ventaja de utilizar un unordered_map sobre un vector se debe por complejidad de procesamiento. Mientras que en este problema el utilizar un vector significa que este ocupe menos memoria posible, esto hará que el proceso sea lento. Por otro lado, el unordered_map causa justo lo opuesto, con esto me refiero que ocupa una gran cantidad de memoria pero el cambio de rapidez de proceso es muy destacado.

Otra duda que nos enfrentamos en la entrega fue en decidir si se implementaría un vector de strings o un struct, el cual ocuparía el segundo elemento del par del unordered_map. Terminamos haciendo un vector de strings ya que no se nos pide ordenar los datos de los ips por estos elementos. Si ese fuera el caso se usaría un struct con los elementos pedidos.

Conclusión Final

Contando cada una de las soluciones a las situaciones problemáticas presentadas, me veo con la necesidad de afirmar que fueron las soluciones de las últimas dos situaciones las que yo considero más eficiente y con un código sin complicaciones, al hablar de estas dos yo me refiero a la implementación de las soluciones utilizando Grafos y Hashmaps. Algo que considero que apoyó mucho en mi razonamiento sobre estas dos actividades fue el descubrimiento de lo que se conoce como unordered_map, lo cual facilitó en gran medida el código a programar, además ayudó con la eficiencia de esta al tener una complejidad constante. Una solución que se pudiera mejorar en su implementación es la actividad sobre la lista doblemente encadenada. La solución se puede mejorar intercambiando la lista

doblemente encadenada ya sea por una implementación de vectores o por una lista encadenada, la cual al compararla con la primera mencionada este última tiene una mayor eficiencia cuando es usada con grandes cantidades de datos, como fue solicitado en la situación problema.

Para finalizar de forma general los conceptos vistos en clase junto con sus actividades impuestas por el profesor estuvieron muy bien estructuradas en mi parecer ya que estas nos ayudaban en entender la implementación de los diferentes temas importantes en la estructura de datos. Algo que me gusto mucho fue la manera que explica el profe y su gusto de enseñarnos cómo no tener un código “naco”, de esta manera mejorando su implementación, presentación y eficiencia del código.