Valeria Conde A01281637 2 de diciembre de 2020

Reflexión final

Binary search

Para realizar el programa, al recibir un archive con datos desorganizados, primero hicimos un sort con la función sort() de la librería de algorithm de C++, que tiene complejidad NlogN. Después creamos una clase usando un struct para crear objetos con cada línea de input y un vector para almacenar los objetos y, finalmente usamos un algoritmo de búsqueda binaria para encontrar los datos que pidiera el usuario, ya que la búsqueda binaria corre en un tiempo logarítmico, por lo que es más eficiente que una búsqueda secuencial, por ejemplo.

La búsqueda binaria la utilizamos una sola vez para encontrar el valor mínimo del rango entre las fechas que se quiere buscar, y después despliega los datos que se piden hasta el valor máximo de la fecha, de manera que no es necesario realizar dos búsquedas binarias y así evitamos llamar dos funciones con complejidad logN.

Finalmente, el programa pregunta al usuario si desea guardar la búsqueda y, en caso positivo, el nombre que desea darle al archivo de salida, y se genera un archivo con los datos ingresados. Esto hace que el programa funcione correctamente, al recibir un archivo, ordenar los datos en él y encontrar cualquiera específico para desplegar una serie de datos al usuario.

Linked lists

Las linked lists son una estructura de datos básica que consiste de nodos encadenados, que componen una lista. La linked list tiene un nodo raíz y se compone de nodos que contienen una información y un apuntador al siguiente nodo. El apuntador del último nodo es un null pointer. Para recorrer una linked list es necesario viajar de nodo en nodo desde la raíz, visitándolos mediante sus apuntadores, por lo que en general la complejidad de recorrerla es de O(n).

Las listas doblemente ligadas, a diferencia de las singly linked lists, pueden ser recorridas en dos direcciones, hacia delante y hacia atrás, y la operación de insertar un nuevo nodo antes de uno existente y otras operaciones son más eficientes, pero cada nodo requiere memoria extra y todas las operaciones requieren un Nodo extra. La complejidad de insertar y borrar un nodo en una lista singly linked es O(n) mientras que en una doblemente ligada es O(1). Además, las listas doblemente ligadas son útiles para implementar stacks, heaps y árboles binarios, mientras que las single no pueden usarse en heaps ni árboles. Este problema se facilita con una doubly linked

list porque permite hacer una búsqueda binaria, recorriendo la lista en ambas direcciones, mientras que una lista single no podría hacerlo.

Binary search tree

El árbol de búsqueda binario es similar a la búsqueda binaria, pero está implementado en forma de árbol, que es similar a la linked lista salvo que éste tiene, en vez de un apuntador a su siguiente nodo, dos, el de izquierda y de derecha del árbol. Esta estructura de datos toma el primer número que recibe y lo usa como raíz del árbol, y el resto de los números que recibe los va enviando a la derecha o a la izquierda de los nodos ya insertados según su valor. De esta manera, mientras se hace inserción de los datos se está creando una especie de orden, y con recorrido del árbol pueden sacarse los valores en orden, y además, reduce la complejidad de la búsqueda a O(log n), aunque en el peor de los casos puede ser lineal. Esto sucedería si los datos que vamos leyendo estuvieran ya ordenados.

Utilizar un BST es util cuando trabajamos con datos en desorden, como en el caso de esta actividad que, a pesar de que la bitacora estaba ordenada, estabamos trabajando con el numero de accesos por IP y por lo tanto se trataba de una randomizacion que hacia posible guardar los datos en esta estructura de datos (esto es importante porque en el caso de que los datos que quisieramos insertar al BST estuvieran en orden el arbol no quedaria balanceado, como es el caso de un AVL o un heap). Usar un BST en esta actividad es util porque queremos encontrar informacion determinada y esa busqueda se reduce a una complejidad de O(log n) cuando esta podria ser mucho mas lenta.

Grafos

Este problema puede ser abordado de distintas maneras y con distintas estructuras de datos. El uso de un grafo ayuda a guardar los datos de manera correcta y permite hacer muchas operaciones con la información que se tiene, como ordenarla con hashing, que además hace que el código sea corto y limpio y por lo tanto fácil de leer y facilita la búsqueda de bugs.

En este problema se hizo un mapeo con una string como llave y un vector como valor, de manera que el uso de <map> de C++ ordenara los datos mientras los leía y esta fue una manera de representar la lista de adyacencias del grafo. Para encontrar los datos que pedía el problema sólo hicimos una iteración del grafo para encontrar sus tamaños y con ello encontrar al botmaster.

El uso del grafo permite tener una estructura ya arreglada con los nodos, por lo que esta es ideal para situaciones así. Los grafos también son ideales para problemas donde existen nodos conectados entre sí, como el mapa de una ciudad con sus calles, por ejemplo.

Hash table

El uso de hashing y hash tables ya implementadas en determinados lenguajes, como <map> y <unordered_map> en C++ son muy útiles para una gran diversidad de problemas, ya que permiten asignar valores a una llave única, y dichas llaves y valores pueden ser de diferentes tipos, por lo que las tablas hash pueden servir como contadores, por ejemplo, si se hashea un valor a otro a otro valor entero, dado que podríamos incrementar el valor del entero en cada aparición de la llave.

En el caso de esta actividad hicimos un mapeo de un valor que representa la dirección de IP a un vector de objetos de tipo un struct que creamos para guardar la información completa que corresponde a ese IP en cada aparición de ese IP mientras se lee el archivo. Es decir, por cada línea que se lee del archivo, a la IP de que se trate, se le asigna un objeto que contiene su fecha y motivo de falla, de tal manera que al finalizar la lectura de los datos la tabla hash contiene **una sola llave** por cada IP diferente y **un vector como valor** que contiene todas las fallas que han ocurrido para esa IP.

Los mapas en C++ corren en O(log n) dado que hacen un sort de las llaves. Como en este caso no nos era necesario tener la información sorteada, se utilizó un “mapa desordenado” que puede correr en complejidad casi constante, lo que lo hace verdaderamente útil y es una de las soluciones más efectivas y rápidas que podemos a darle al problema. En general, las tablas de hash son muy útiles y pueden usarse para resolver muchísimos tipos diferentes de problemas.