**Reflexión Actividad 3.4**

Cristóbal Alberto Escamilla Sada - A00827074

A través de la actividad 3.4 realizamos la lectura de un archivo de entrada conteniendo registros de accesos fallidos a diferentes redes. A partir de esta lectura colocamos las diferentes direcciones IP en un árbol binario de búsqueda organizado en orden descendente por cantidad de accesos. Es decir, la dirección IP con más repeticiones en el archivo es la de mayor valor en el árbol (hoja en el extremo derecho). Para esto no se consideraron los puertos de las direcciones IP.

La clase NodeT está conformada por 4 atributos. ip es de tipo string y contiene la dirección ip sin considerar el puerto. key es de tipo entero y almacena la cantidad de veces que se repite dicha dirección ip en la bitácora de registros. Por último, left y right son apuntadores a un objeto de tipo NodeT que contienen la dirección del nodo a la izquierda y a la derecha, respectivamente. En la sección privada de la clase también se encuentra la función ipToLong. Esta se declara como privada ya que solamente se utiliza dentro de la misma clase NodeT. Este se encarga de convertir la dirección IP de tipo string a tipo long para facilitar la comparación entre los nodos. Tiene una complejidad de O(n) debido a que recorre cada elemento del string para posteriormente convertirlo a un long int. La clase tiene funciones *setters* y *getters* para cada uno de los atributas; todas de complejidad constante. También se sobrecarga el operador *mayor que*— > con el propósito de facilitar la comparación entre la dirección IP de 'x' nodo con una nueva dirección ip de entrada. Esta sobrecarga toma como parámetro una dirección IP de tipo string y la compara con la dirección IP almacenada en el nodo llamando a la función ipToLong. Por esta razón, la sobrecarga tiene una complejidad de O(n). Finalmente se realizó la sobrecarga del operador de inserción— << para desplegar los datos del nodo. Al llamar a este operador, se despliega la dirección IP así como su cantidad de accesos. Esta tiene una complejidad constante.

La clase BST es la que forma el árbol binario de búsqueda con nodos de tipo NodeT. El único atributo de un objeto BST es un apuntador de tipo NodeT llamado root. Este apunta a la dirección en memoria que almacena el nodo raíz del árbol. La función add se encarga de colocar los nodos en el árbol de forma descendente por cantidad de accesos a la dirección IP. En caso de que las cantidades de accesos de dos nodos sean iguales, el segundo criterio es el valor de la dirección. Esto se hace en orden ascendente. Es decir, lo primero a considerar es la cantidad de accesos y en caso de que sean las mismas se considera el valor de la dirección IP. Esto significa que la hoja más a la derecha del árbol contiene la dirección con el mayor número de accesos y menor valor de IP dentro de esta cantidad de accesos. add tiene dos parámetros de entrada: int key y string ip. En el peor caso, la función add tiene una complejidad de O(n^2) ya que el while loop itera n veces; n siendo la cantidad de niveles en el árbol e implementa la función ipToLong en caso de que el número de accesos se repita. La función inordenConv se encarga de desplegar los nodos del árbol en inorden converso. Es decir, el primer dato que va a desplegar es el que se ubica más a la derecha. Esta funciona de manera recursiva y tiene una complejidad de O(n). Se declara como privada ya que posteriormente será utilizada en la función pública print para realizar el despliegue en pantalla. La función privada printFive se implementa dentro de inordenConv para controlar la cantidad de despliegues que se hacen en pantalla. En este caso solamente se desplegarán las primeras cinco direcciones y se determina a través de un contador. Tiene una complejidad constante. La función pública print llama directamente a inordenConv para desplegar los datos, tiene una complejidad lineal.

Finalmente la función destruye se encarga de eliminar todos los nodos del árbol y se implementa en el destructor de la clase. Tiene una complejidad lineal.

En el archivo main.cpp se leen los datos de entrada y se almacenan en un BST. La función cargaRegistros recorre todo el archivo de entrada y llama a la función add de BST una sola vez por dirección IP. Debido a que add es de complejidad cuadrada, la función cargaRegistros tiene una complejidad de O(n^3) en el peor caso.

La estructura de datos BST es sumamente eficiente para este tipo de problemas. Es fácil insertar los datos en el árbol binario y también es fácil desplegarlos en diferentes órdenes dependiendo del proyecto. Los datos se ordenan de una manera eficiente y de esta manera los modos de acceso son menos complejos que los de una lista encadenada. Se puede determinar si una red está infectada dependiendo de su cantidad de accesos fallidos. En caso de que una dirección IP tenga un número alto de accesos fallidos se podría decir que esta intenta ser *hackeada*. Con esto se llega a la conclusión de que la red más infectada de la bitácora es la 10.15.176.241 con 38 accesos fallidos. Los administradores de la red deben de tomar los cuidados necesarios para evitar el robo de su información privada. Este es un problema que ocurre cada minuto en la industria tecnológica y el BST es una gran solución para el almacenamiento así como el ordenamiento de las direcciones.