Reflexión Actividad 2.3

Cristóbal Alberto Escamilla Sada - A00827074

El objetivo de esta actividad fue ordenar un archivo de entrada “bitacora.txt” conteniendo más de 16,000 datos, buscar y desplegar el rango de datos proporcionado por el usuario (rango de direcciones IP), así como transferir los datos ordenados a un archivo nuevo. Para lograr esto, lo primero que se consideró fue el formato del archivo bitacora.txt. Este está organizado por renglones, cada uno conteniendo un registro diferente. Cada renglón cuenta con 5 atributos: mes, día, hora, dirección IP, y razón de falla. Para facilitar el uso de vectores decidimos crear una clase “Registro.h” con dichos atributos. Adicionalmente, implementamos la función ipToLong la cual fue proporcionada por el profesor para facilitar la comparación de las direcciones IP. Debido a que estas son de tipo string, a través del ipToLong estas direcciones se convierten al tipo de dato long int y de esta manera se pueden ordenar de forma ascendente. Esta función tiene una complejidad lineal, O(n) debido a que recorre cada uno de los caracteres en la dirección IP para posteriormente convertirlos en long ints. ipToLong fue declarada como privada debido a que es llamada en diversas funciones dentro de la misma clase. Esto se hizo con eficiencia para no tener que reescribir los contenidos en la función. Posteriormente elaboramos la función pública convierteIP. Esta llama a la función ipToLong usando el atributo direccionIP como parámetro. Tiene una complejidad de O(n) debido a que llama a una función lineal. También se sobrecargaron los operadores “mayor o igual que >=” y “menor o igual que <=” para comparar las direcciones IP de los registros con las direcciones IP ingresadas por el usuario. Estas sobrecargas tienen una complejidad lineal, O(n) debido a que llaman a la función ipToLong.

Los registros de la bitácora fueron almacenados a través de un tipo de dato abstracto. En este caso se usó una lista doblemente encadenada conteniendo nodos con la información de los registros. Una ventaja que tiene esta estructura de datos es que por ser doblemente encadenada, puede ser accesada a través del apuntador head— conteniendo la dirección de memoria del primer nodo en la lista, así como a través del apuntador tail— conteniendo la dirección de memoria del último nodo en la lista. A parte de esto, los nodos están doblemente encadenados entre ellos. Es decir, a partir de un nodo se tiene acceso al nodo anterior así como al siguiente nodo.

La clase Node tiene tres atributos: Uno de tipo Registro que se encarga de guardar el registro correspondiente al nodo (data), un apuntador de tipo nodo que almacena la dirección del siguiente nodo en la lista (next) y otro apuntador de tipo nodo que almacena la dirección del nodo anterior en la lista (prev). Esta clase tiene getters y setters para todos los atributos así como una sobrecarga del operador “mayor que >” cuyo propósito es comparar la dirección IP del registro almacenado en el nodo con otra dirección IP almacenada en un nodo distinto. Esta sobrecarga tiene una complejidad de O(n) debido a que llama a la función convierteIP de la clase Registro.

La clase DoubleLinkedList es la que forma la estructura de datos uniendo los Nodos que contienen los registros. Este tipo de dato tiene tres atributos: un apuntador de tipo nodo que almacena la dirección del primer nodo de la lista (head), otro apuntador de tipo nodo que almacena la dirección del último nodo de la lista (tail) y un entero que almacena el tamaño de la lista (size). Por default, la lista vacía inicia con el head y el tail teniendo valores de nullptr. Las funciones addFirst y addLast se encargan de agregar nodos al principio o al final de la lista. Ambas tienen una complejidad constante, O(1) debido a que los nodos se pueden agregar a través del acceso de los apuntadores head y tail. La función sortIP es la

que se encarga de ordenar los nodos de forma ascendente tomando en cuenta las direcciones IP de los registros almacenados. Se usó un ordenamiento de tipo burbuja debido a las limitaciones en el acceso de los nodos. Las comparaciones se realizan de dos en dos nodos, en dos *while loops* anidados con órdenes lineales. Debido a que en las comparaciones se usa las sobrecargas de > cuyo orden es lineal, la función termina teniendo una complejidad de orden cúbico, O(n^3). Este es un método sumamente ineficiente ya que el ordenar 16,000 datos resulta en una gran demora. Finalmente, se realizó la función busqueda cuyo propósito es encontrar los datos dentro del rango de direcciones IP establecido por el usuario. La búsqueda se hace secuencialmente debido a las limitaciones de acceso en los nodos. Debido a que las comparaciones se realizan mediante las sobrecargas de >= y <= cuyos órdenes son lineales, la función resulta tener una complejidad de orden cuadrado, O(n^2); también sumamente ineficiente. Por último se realizó la sobrecarga del operador de inserción para imprimir todos los datos que contiene la lista doblemente encadenada.