Tecnológico de Monterrey

Jose Manuel Martinez

A01734279

26 de julio de 2023

Actividad Integral de estructura de datos lineales

programación de estructuras de datos y algoritmos fundamentales

Jorge Enrique González Zapata

Análisis de complejidad

insert(Node* node, const string& key, const string& ip)

Esta función es utilizada para añadir un nuevo nodo al árbol los cuales llevan la key que son el número de veces que aparece un puerto en bitácora y su ip asociada, compara la key que se quiere insertar con el nodo actual y dependiendo si es menor lo inserta en el subárbol izquierdo y si es mayor en el subárbol derecho, hasta que encuentre una posición. Si ese valor ya existe en el árbol se le suma uno al contador de ese puerto y se le asocia la IP. La complejidad puede ser O(log n) en un árbol de búsqueda binaria balanceada, pero puede ser O(n) para un árbol sesgado, el cual es el peor caso.

topNPortsHelper(Node* node, vector<pair<string, int>> sortedPorts, int N)

Esta función es auxiliar de getTopNPorts, nos ayuda a encontrar los puertos que tengan una mayor numero de búsqueda gracias a un recorrido inverso, que primero recorre todo el subárbol derecho, luego la raiz y finalmente el subárbol izquierdo, de igual manera guarda en un vector el número de veces que se repiten, así como el puerto. La complejidad puede ser O(log n) en un árbol de búsqueda binaria balanceada, pero puede ser O(n) para un árbol sesgado, el cual es el peor caso.

findNode(Node* node, const string& port)

Esta función busca un nodo con un puerto que queramos en el árbol utilizando una búsqueda binaria. La complejidad puede ser O(log n) en un árbol de búsqueda binaria balanceada, pero puede ser O(n) para un árbol sesgado, el cual es el peor caso.

getTopNPorts(int N)

Esta función nos devuelve un vector con el top que deseamos de los puertos más accesos, así como las veces que se repiten utilizando la funcion getTopNPortsHelper. Por lo que la complejidad puede ser O(log n) en un árbol de búsqueda binaria balanceada, pero puede ser O(n) para un árbol sesgado, el cual es el peor caso.

printIPsForPort(const string port)

Esta función imprime las Ips asociadas a un puerto especifico y llama a findNode para localizar el nodo del puerto. La complejidad puede ser O(log n) en un árbol de búsqueda binaria balanceada, pero puede ser O(n) para un árbol sesgado, el cual es el peor caso.

buildBSTFromFile(const string filename)

Esta función construye un árbol BST a partir de un archivo, lee todas las linas del archivo y extrae el puerto y la dirección IP y llama a insert para agregarlos al árbol, la complejidad de esta función sola es O(n) porque lee las líneas del archivo, pero como manda a llamar a insert tenemos que su complejidad puede ser de O(n log n) si es un árbol balanceado o O(n^2) si es un árbol sesgad que es el peor caso.

Reflexión

La utilización de un Binary Search Tree en la detección de infecciones de una red puede ser útil a la hora de identificar patrones que sean anormales en Ips, lo cual puede indicar que hay alguna amenaza en la red, en este caso se realizó un código para poder detectar las veces en las cuales un puerto tenía un mayor número de accesos. Con la ayuda del BST pudimos acomodar los nodos de forma que los puertos que tuvieran más accesos fueran ubicados en el subárbol derecho para poder identificar a través de un postorden las anormales ips que se registraban en esos puertos.

La implementación de un BST nos fue beneficiosa en este caso de detección de infecciones al tener una búsqueda eficiente para identificar las direcciones IP y puestos que tengan registros sospechosos. De igual manera la implementación de un BST me permitió organizar los datos de manera ordenada para facilitarme a identificar patrones o comportamientos en los puertos de la bitácora, al poner los puertos con menores accesos en un subárbol izquierdo y los de más accesos en un subárbol derecho para tenerlos clasificados de mejor manera. Otra forma en la que un BST fue de utilidad en esta situación problema fue a la hora de identificar los puertos que tenían más repeticiones para poder detectar actividad sospechosa.

Sin embargo el utilizar un BST tiene alguna desventaja como puede ser el balance del árbol ya que si el árbol esta balanceado presenta en algunas de sus funciones clave una complejidad de

O(log n), pero si árbol se encuentra desbalanceado o sesgado se convierte en una lista ligada lo que hace que esas funciones tenga unan complejidad de O(n) lo que disminuye la eficiencia.

En conclusión, la implementación del BST proporcionó una estructura de datos organizada y una búsqueda rápida para identificar patrones anormales en los puertos y direcciones IP registradas en la bitácora. Aunque el BST puede tener desafíos en su balance, con esto podríamos implementar técnicas de detección si hay un numero anormal de registros y proteger la red de posibles amenazas.