Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Programación de estructuras de datos y algoritmos fundamentales – TC1031

Act 3.4 - Actividad Integral de BST (Evidencia Competencia)

Reflexión personal

Grupo 14

Adrián Alejandro Salgado Martínez – A00828843

25 de octubre de 2020

Estructuras de datos jerárquicas - Reflexión

Los Binary Search Trees (BST), también conocidos como árboles de búsqueda binaria, son un tipo de estructura de datos que poseen características estructurales y funcionalidad que es de gran utilidad para algunas situaciones donde se requiere de una búsqueda eficiente dentro de sus datos, puesto a que por definición por cada nodo del árbol existe un subárbol donde todos los elementos a la izquierda son menores y a la derecha son mayores, logrando así una más rápida búsqueda al descartar la mitad de los datos en cada iteración para un árbol bien balanceado, es decir, uno donde la diferencia de altura de los subárboles no sea mayor a 1.

Con esto en mente, para darle solución a la situación problema planteada que consiste en encontrar las IPs con un mayor número de accesos por medio de un archivo de texto, se implementó una clase contenedora con el número de accesos y la IP, además de un BST y un iterador constante (para solo leer la información) y reverso (para recorrer de mayor a menor) donde los principales algoritmos utilizados fueron los siguientes:

- `BST<Registro>::add(Registro)` – Este método del árbol es el encargado de añadir al mismo el nuevo elemento que se desea guardar. Esto lo hace comparando al dato nuevo que se desea insertar con los distintos nodos, y haciendo el recorrido necesario hasta poder insertarlo como nuevo nodo hoja. Dependiendo del balance del árbol esta operación se puede llevar de manera lineal como se haría con una `Linked List` (todos a la derecha o todos a la izquierda) con complejidad de $O(n)$, y en el caso de un árbol balanceado sería de $O(\log n)$, por lo que se dice que en realidad la complejidad dependerá de la altura del árbol, en otras palabras será de $O(h)$.
- `BST<Registro>::ConstReverseIterator::operator++()` – Este método del iterador realmente es el más importante del mismo, puesto a que es el que se encarga de partir del nodo al que pertenece el iterador y encontrar el siguiente nodo siguiendo una secuencia transversal 'InOrder' conversa. La manera en la que se realiza esto es que se determina si es que el nodo al que pertenece el iterador tiene un hijo izquierdo, si es así se busca su predecesor y si no se mueve hacia arriba en el árbol hasta encontrar el siguiente en este orden. Para conseguir esto último, una de las maneras posibles de implementarlo es por medio de un stack que mantenga el recorrido desde la raíz hasta el nodo actual, sin embargo, se decidió que para simplificar las cosas cada nodo tendría además un atributo que apunte al nodo padre del mismo. Finalmente, al igual que la operación pasada este depende completamente de si el árbol se encuentra balanceado o no, por lo que tiene una complejidad de $O(h)$.
- `BST<Registro>::rend()` y `BST<Registro>::rbegin()` – Estos dos métodos son los más simples de la clase. El primero de ellos navega el árbol hacia la derecha y una vez llega al nodo mayor regresa un iterador con este nodo, y al igual que los anteriores es dependiente de si se encuentra balanceado por lo que tiene una complejidad de $O(h)$. El segundo solo regresa un iterador con nullptr, con esto se logra una complejidad $O(1)$.

A partir de estos algoritmos se logró obtener entonces estas IPs con más accesos, que en el contexto de la problemática planteada nos permitiría determinar la posibilidad de que un sistema esté infectado, pues al observar de qué IPs provienen el mayor número de accesos fallidos en un corto tiempo se puede determinar cuál es el sistema con más probabilidad de estar infectado y actuando como un 'bot' en una 'botnet', asimismo al observar que una IP tiene un bajo número de accesos fallidos cercanos se puede concluir que es más probable que en realidad se trate de simplemente de una persona o de la mala configuración del servidor.

Finalmente, se concluye que en este caso particular el uso de un BST tuvo importantes ventajas. Principalmente que este inherentemente mantiene una estructura ordenada por lo que no es necesario recorrer todas las entradas de nuevo para ordenarlas, si no que al irlas agregando estas ya estarán en orden. No obstante, se reconoce que el BST que se implementó es el más sencillo que puede haber, careciendo de la habilidad de otro tipo de BSTs más especializados de balancearse a sí mismo, que a cambio de un poco más de complejidad computacional al insertar o buscar elementos traen consigo la ventaja de garantizar este balance. Asimismo, se observa que a pesar de la facilidad del ordenamiento de los datos, realmente acceder de manera secuencial a los mismos es considerablemente más complicado que con las estructuras de datos con las que se trabajó anteriormente, por lo que este tipo de datos son de mejor uso para búsquedas irregulares de elementos no secuenciales.