Tecnológico de Monterrey

Act 3.3 - Actividad Integral de Binary Search Tree BST (Evidencia Competencia)

Nikole Morales Rosas - A01782557

TECNOLÓGICO DE MONTERREY

Campus Santa Fe

Grupo: 570

Fecha de entrega: 22 de julio de 2023

Profesor: Dr. Eduardo A. Rodríguez Tello

Programación de estructuras de datos y algoritmos fundamentales

Binary Search Tree

Reflexión

Las estructuras de datos jerárquicas son de suma importancia para poder ordenar datos, gracias a estas estructuras se puede representar una relación “padre-hijo” en forma de árbol, al estar conectados los datos de manera jerarquizada es más fácil tanto acceder a ella como también a poder organizarla.

En este caso en particular una estructura jerarquizada permite acceder a los datos a través de niveles, lo que facilita la búsqueda y el análisis de registros relacionados con una misma IP. Así mismo se agrupa la información para detectar actividades repetitivas e identificar si una misma red intenta ingresar varias veces.

Para esta actividad se utilizó una de estas estructuras de datos de árbol general BST (Binary Search Tree) y un algoritmo HeapSort para ordenar y almacenar los registros de una bitácora.

Según se explica en un artículo de la UAM -“El algoritmo HeapSort, consiste en remover el mayor elemento que es siempre la raíz del Heap, una vez seleccionado el máximo, lo intercambiamos con el último elemento del vector, decrementamos la cantidad de elementos del Heap y nos encargamos de reacomodarlo para que vuelva a ser un Heap.” (UAM, s/f).

En cuanto a la construcción y ordenamiento del Heap dentro del programa implementado, la construcción inicial del Max Heap se realiza en el bucle for en la función heapSort, donde se ejecuta “heapifyHelper” para cada nodo que no sea hoja en el árbol, comenzando desde el último nodo interno y retrocediendo hacia el primer nodo. En cada llamada a heapifyHelper, la función se mueve hacia abajo a través del árbol, ajustando los nodos para mantener la propiedad del Max Heap.

Después de la construcción del Max Heap, se realiza la fase de ordenación real. Se extrae el elemento máximo (la raíz) del Heap y se coloca en la última posición del vector, y luego se invoca “heapifyHelper” para mantener la propiedad del Max Heap en el subárbol restante (sin incluir el elemento extraído).

Con este algoritmo se logró ordenar los registro mediante su IP para posteriormente, a partir de la bitácora ordenada se contabilice la cantidad de accesos de cada IP esto se utilizó una estructura de datos “unordered_map” para almacenar las direcciones IP como claves y el número de accesos correspondiente como valores.

Basado en el artículo de GeekForGeeks se implemento el “unordered_map” que se implementó para realizar un recuento eficiente del número de accesos para cada dirección IP almacenada en el vector de objetos Registro.

```
void AccessTracker::cuentaAccesos(const std::vector<Registro>& _vectorReg){
    std::unordered_map<std::string, int> contador;
    unsigned int comparacion, intercambio;

    for (const Registro& registro : _vectorReg){
        contador[registro.getIp()]++;
    }

    MaxHeap<AccessTracker> heapIp(contador.size());

    for (const auto& par : contador){
        heapIp.push(AccessTracker(par.first, par.second));
    }

    heapIp.saveHeap("contabilizacion_Ips_heap.txt");

    std::cout << "Se almaceno correctamente en un Heap Tree los IPs junto con su numero de
apariciones en la bitacora." << std::endl;
    std::cout << "Consulte en el archivo 'contabilizacion_Ips_heap.txt'." << std::endl;

    obtieneMayorIp(heapIp, "ips_con_mayor_acceso.txt");
}
```

Fig 1.1: Sección del código donde se utiliza la estructura de datos “unordered_map”

Para contar las veces que aparece cada dirección IP en la bitácora, se utiliza la estructura de datos “unordered_map” como se muestra en la *Figura 1.1*. Esta estructura almacena las direcciones IP como claves y el número de accesos correspondiente como valores. De esta manera, se puede realizar un recuento eficiente de los accesos para cada dirección IP.

La función “cuentaAccesos” realiza el proceso de contar los accesos y llenar el “unordered_map”. Primero, se itera sobre el vector de objetos Registro, y para cada objeto, se aumenta el valor asociado a la clave correspondiente en el “unordered_map”. Después de esto se crea un Max Heap con el “unordered_map” una vez que se completó.

La función "cuentaAccesos" itera sobre el vector de Registro y aumenta el valor asociado a la clave correspondiente en el "unordered_map", contabilizando los accesos, mientras que va llenando el contenedor. Luego, se crea un Max Heap basado en el "unordered_map" para encontrar las 10 direcciones IP con mayor número de accesos.

Finalmente, se guardan estas direcciones IP en un archivo llamado "**ips_con_mayor_acceso.txt**", esto gracias al método “obtineMayorIp” (esta función toma el Max Heap “heapIp” y extrae los 10 elementos principales del Heap) .

En cuanto a la complejidad computacional total del algoritmo de HeapSort es la suma de las complejidades de construcción y ordenación del Heap que es O(n log n). Mientras que la complejidad global del programa depende del tamaño del vector "vectorRegistros" y del Max Heap. Si el número de objetos en el vector es mayor que 10, la complejidad dominante sería O(n) debido a la función "cuentaAccesos", mientras que si el tamaño de "heapIp" es grande, la complejidad dominante sería O(log n) debido a la función "obtieneMayorIp"

Con lo anterior nos podemos dar cuenta de que esta situación problema tiene el objetivo de implementar un programa que permite realizar un análisis de la bitácora de acceso de una red sobre la cantidad de veces que una misma IP ha intentado ingresar y los mensajes de error y/o falla. Con esto se puede determinar si una red está infectada ya que cuando una IP intenta repetidamente ingresar a un red en un lapso de tiempo corto, se considera una actividad inusual o maliciosa. Por eso gracias a los métodos implementados se puede recopilar la información mencionada anteriormente sobre los intentos de accesos de IP's y determinar si existe algún comportamiento anómalo para saber si la red está infectada.

En conclusión, en un programa que ordena una bitácora mediante IP´s para detectar intentos repetidos de ingreso, una estructura de datos jerárquica como un árbol podría ser útil. Por ejemplo, se podría usar un BST en el que cada nodo contenga una dirección IP y la información asociada a esa IP (como la fecha y la hora de cada intento de acceso). De esta manera, los intentos repetidos de una misma IP estarían agrupados en el mismo subárbol, lo que facilita su detección y análisis.

Referencias

Heap sort - data structures and algorithms tutorials. (2013, marzo 16). GeeksforGeeks. Recuperado el 22 de julio de 2023 en: https://www.geeksforgeeks.org/heap-sort/

Bases de datos jerárquicas ¿Qué son? Ejemplos. (2020, septiembre 9). Ayuda Ley Protección Datos; AyudaLeyProteccionDatos. Recuperado el 22 de julio de 2023 en: https://ayudaleyprotecciondatos.es/bases-de-datos/jerarquicas/

Unordered_map in C++ STL. (2016, marzo 24). GeeksforGeeks. Recuperado el 22 de julio de 2023 en: https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/

Anónimo. std::unordered_map. (s/f). Cplusplus.com. Recuperado el 22 de julio de 2023 en: https://cplusplus.com/reference/unordered_map/unordered_map/