Reflexión Actividad Integradora 3

Gonzalo Flores García - A01644771

Analiza la importancia y eficiencia del uso de estructuras de datos jerárquicas en una situación problema de esta naturaleza.

Este caso tuvo una necesidad mayor de usar una estructura de datos jerárquica debido a que se nos pidió acceder a los elementos de mayor y menor prioridad. A diferencia de tener que recorrer un arreglo, la estructura del binary heap nos permite reducir muchísimo los tiempos de búsqueda para lo que se nos pedía. Eso combinado con el hecho de que la inserción es de complejidad logarítmica, termina resultando en una muy buena estructura de datos para problemas con volúmenes tan grandes.

Reflexiona por qué en la solución de este reto es preferible emplear un Binary Heap y no un BST, basa tu reflexión en la complejidad temporal de las operaciones de ambas estructuras de datos.

Primero que nada, esto es asumiendo que nuestro BST es balanceado, pero la inserción de este y la búsqueda es de complejidad O(log n). Esto nos podría dar la falsa sensación de que sería lo mismo aplicar un árbol binario y un binary heap, pero la realidad es que el segundo nos conviene mucho más solamente porque se nos pide resolver dudas de prioridad. Mientras que sacar el máximo o el mínimo en un BST también es de complejidad logarítmica, hacer esto en un binary heap es de O(1). Es por esto que se dice que es una estructura de datos especializada en prioridad.

Analiza la complejidad computacional de las operaciones básicas de la estructura de datos empleada en tu implementación (push, pop, getTop, etc) y cómo esto impacta en el desempeño de tu solución

La implementación que nosotros realizamos tiene complejidades O(log n) en las partes de la inserción y el heapify. La complejidad más alta a la que llegamos es de O(n log n) con nuestro acomodo heapSort() pero por fortuna eso solo se debe hacer al inicio. Por otra parte, conseguir nuestro número de repeticiones por ip para poder aplicar nuestro heap con esta prioridad es de O(n), haciendo que por último nuestras funciones para encontrar las ip con más ingresos sea O(m log m) y encontrar las menos que fueran mayores o iguales a 3 O(m). Esto tomando en cuenta que m es el número de ip 's únicas. Viendo nuestras

complejidades creo que podemos llegar con seguridad a la conclusión de que nuestra implementación es eficiente para volúmenes altos de datos.

Presenta el pseudo-código del algoritmo empleado para buscar la IP con menor número accesos en un Binary Heap (del punto 6), así como el análisis de su complejidad temporal.

Pseudocódigo

```
function findLeastRecurringIpAboveThreshold(threshold):

    for each (ip, count) in ipAccessCount:
        if count ≥ threshold and count < minCount:
            minCount ← count
            leastRecurringIp ← ip

    if leastRecurringIp is empty:
        return ("", 0) #No existe

    return (leastRecurringIp, minCount)
```

En nuestro caso no implementamos un min-heap para esta operación y en lugar usamos un mapa de la ip y su número de repeticiones. Esto es lo que nos lleva a nuestra complejidad de O(m), m siendo el número de ip distintas.

Bibliografía

GeeksforGeeks. (2025, febrero 14). *Introduction to Min-Heap – Data Structure and Algorithm Tutorials*. Recuperado de https://www.geeksforgeeks.org/introduction-to-min-heap-data-structure/?ref=rp

GeeksforGeeks. (2024). *Difference between Binary Search Tree and Binary Heap*. Recuperado de https://www.geeksforgeeks.org/difference-between-binary-search-tree-and-binary-heap/

Wikipedia. (2024). *Binary heap*. Recuperado de https://en.wikipedia.org/wiki/Binary_heap

Stack Overflow. (2011). *Heap vs Binary Search Tree (BST)*. Recuperado de https://stackoverflow.com/questions/6147242/heap-vs-binary-search-tree-bst