Tecnológico de Monterrey

Nombre: Rodrigo Castillo Francisco

Matricula: A01799191

Materia: Programación de estructuras de datos y algoritmos fundamentales

Profesor: Eduardo Arturo Rodríguez Tello

1. Importancia y eficiencia del uso de estructuras jerárquicas

En problemas que son de gran escala como el trabajo que se realizo que es el análisis de bitácoras de red, es importante poder extraer con rapidez los elementos extremos como por ejemplo las IPs mas frecuentes a la menos frecuente bajo cierta condición sin recorrer rápidamente toda la lista. Las estructuras jerárquicas, como lo que se utilizo que fueron los heaps binarios, ofrecen esta capacidad almacenando los datos en un vector casi completo y garantizando que la raíz siempre contenga el elemento máximo o mínimo gracias a las operaciones de subir y bajar nodos para poder restaurar la propiedad de heap, además se puede insertar y extraer el elemento extremos en un tiempo logarítmico logrando un rendimiento que sea escalable cuando n es muy grande.

2. Por qué emplear Binary Heap y no BST

En mi opinión, aunque hacer un árbol de búsqueda balanceado (BST) también permite hacer inserción y extracción, en la práctica los heaps binarios tienen constantes mucho menores y un código que es realmente más sencillo. Un BST requiere rotaciones tras cada inserción o cuando se usa el borrado para conservar su altura mínima, lo que agrega coste a cada operación. Un heap solo intercambia nodos a lo largo de un solo camino sin necesidad de mantener punteros de padre o de recalcular factores de balance. Esto hace que el heap sea mucho mas ligero y rápido de implementar para colas de prioridad donde solo interesa el máximo o mínimo.

3. Complejidad computacional de las operaciones básicas

Push(value): Inserta el par (conteo, IP) al final del vector y lo sube hasta restaurar el heap en O (log u), donde u es el numero de IPs que son únicas.

pop (): Elimina la raíz sin modificar la estructura en O (1).

top (): Devuelve la raíz sin modificar la estructura en O(1).

isEmpty () y size() : Acceden directamente al vector subyacente en O(1). Este comportamiento garantiza un procesamiento eficiente incluso con millones de registros.

4. Pseudocódigo para encontrar la IP con menos acceso >=3

Pseudocódigo

```
1.  Si longitud(A) = 0 Entonces
2.      retornar ("", 0)
3.  FinSi
4.  inicio ← floor(longitud(A) / 2) // índice de la primera hoja
5.  bestCount ← longitud(A) + 1 // centinela mayor que cualquier conteo
6.  bestIP ← ""
7.  Para i ← inicio Hasta longitud(A) − 1 Hacer
8.      count ← A[i].first
9.      ip ← A[i].second
10.     Si count ≥ minCount Entonces
11.         Si count < bestCount Entonces
12.             bestCount ← count
13.             bestIP ← ip
14.         FinSi
15.     FinSi
16. FinPara
17. Si bestCount = longitud(A) + 1 Entonces
18.     retornar ("", 0)
19. FinSi
20. retornar (bestIP, bestCount)
```

5. Conclusion

Este programa demuestra como elegir estructuras de datos adecuadas en este caso Binary Heap marca una gran diferencia en eficiencia y claridad. Al usar Heap Sort para ordenar las IPs y un MaxHeap para contabilizar y extraer rápidamente las diez IPs más frecuentes, minimizando el tiempo de procesamiento incluso con volúmenes con muchísimos registros. La técnica de

escaneo directo de las hojas para encontrar la IP con menor número de accesos en O(u) en donde evita costos extras de memoria y de construcción de un segundo heap, mostrando como pequeños ajustes algorítmicos pueden optimizar el rendimiento. En conjunto, el diseño modular y el análisis de complejidad aseguran que el código sea escalable, fácil de mantener y perfectamente alineado con los objetivos de la actividad.