ReflexAct5.2 Marco

Marco Antonio García Mendoza a01026487

En este problema, donde almacenamos datos de direcciones IP junto con su cantidad de salidas y entradas, las tablas hash son esenciales debido a lo rápidas que son para buscar y obtener información.

Imaginemos que las tablas hash son como casilleros con una magia especial para encontrar rápidamente lo que necesitas. Con una fórmula mágica, podemos convertir direcciones IP en números y guardarlas en estos casilleros. Si la fórmula es buena, podemos encontrar la información en tiempo súper rápido, sin importar cuánta información haya.

Pero a veces, dos direcciones IP pueden convertirse en el mismo número y quedar en el mismo casillero. Eso se llama una colisión. Es como cuando dos personas tienen el mismo número de casa. Para resolver eso, usamos trucos como probar casilleros cercanos hasta encontrar uno vacío.

La rapidez de las operaciones en la tabla hash depende de qué tan buena sea la fórmula mágica y de cómo manejemos las colisiones. En general, agregar, quitar y buscar datos en una tabla hash es súper rápido, pero si hay muchas colisiones, puede volverse más lento. Por eso es importante tener una buena fórmula y manejar bien las colisiones. Si hay muchas colisiones, la búsqueda y recuperación de datos pueden volverse más lentas, lo que afecta el rendimiento general del sistema. Por eso, debemos asegurarnos de que nuestra fórmula distribuya uniformemente los datos y de que nuestras técnicas para manejar las colisiones sean eficientes.

Complejidades computacionales:

hash(std::string ip): Complejidad: O(n), donde n es la longitud de la cadena ip. Esto se debe a que esta función recorre cada carácter de la cadena para calcular la suma de los valores ASCII.

hashWithQuadraticProbing(std::string ip, int attempt): Complejidad: O(1) en promedio. Aunque el método de resolución de colisiones utiliza un enfoque cuadrático, el tamaño de la tabla hash es constante, lo que hace que el tiempo de ejecución promedio sea constante.

insert(IPInfo* ipInfo): Complejidad: O(1) en el mejor caso y O(n) en el peor caso, donde n es el tamaño de la tabla hash. En el mejor caso, la inserción se realiza en una posición vacía de la tabla hash sin colisiones. En el peor caso, todas las posiciones de la tabla están ocupadas, lo que requiere una búsqueda lineal completa para encontrar un espacio vacío.

getInfo(std::string ip): Complejidad: O(1) en el mejor caso y O(n) en el peor caso, donde n es el tamaño de la tabla hash. En el mejor caso, la IP buscada se encuentra en la posición calculada por la función hash sin colisiones. En el peor caso, se deben recorrer todas las posiciones de la tabla hash debido a colisiones.

getTotalCollisions(): Complejidad: O(1). Este método simplemente devuelve el valor de una variable miembro, por lo que su complejidad es constante.

Carga del grafo desde un archivo: O(V + E), donde V es el número de vértices y E es el número de aristas en el grafo. Esto se debe a que debemos leer cada línea del archivo una vez para construir la representación del grafo.

Shortest Path: El algoritmo Dijkstra es comúnmente utilizado para calcular el camino más corto en grafos ponderados. Complejidad temporal: O((V + E) * log V) utilizando una cola de prioridad implementada con un heap binario.

Boot Master: Para encontrar el vértice con el mayor grado, necesitamos calcular el grado de cada vértice y luego identificar el que tenga el mayor grado. Complejidad: O(V + E).
Max Heap: La construcción de un montículo máximo a partir de una lista de elementos toma tiempo lineal. Complejidad temporal: O(n).
Bubble sort: compara repetidamente cada par de elementos adyacentes y los intercambia si están en el orden incorrecto. Complejidad: O(n^2)
Insertion Sort: construye una lista ordenada de elementos uno por uno, tomando cada elemento y colocándolo en la posición correcta en la lista ordenada. Complejidad: O(n^2)
Merge Sort: Divide recursivamente la lista en mitades más pequeñas, ordena esas mitades y luego las fusiona para obtener una lista ordenada. Complejidad: O(n log n)
Quick Sort: elige un elemento como pivote y particiona la lista alrededor del pivote, de modo que los elementos menores que el pivote estén a su izquierda y los elementos mayores estén a su derecha. Complejidad: peor caso O(n^2) y su mejor caso O(n log n)
Búsqueda Lineal: busca secuencialmente cada elemento en una lista hasta encontrar el elemento buscado o llegar al final de la lista. Complejidad: O(n)
Búsqueda Binaria: Compara el elemento buscado con el elemento en la mitad de la lista y descarta la mitad no necesaria del arreglo en cada paso. Complejidad: O(log n)
Búsqueda en árboles binarios de búsqueda (BST): utiliza la estructura de un árbol binario de búsqueda, donde los nodos están organizados de manera que los elementos menores están a la izquierda y los elementos mayores están a la derecha. Complejidad: En promedio O(log n), pero en el peor caso podría ser de O(n) si el árbol no está balanceado.

Referencias:
Hash Collisions Explained. (2022, October 4). Freeman Law. https://freemanlaw.com/hash-collisions-explained/
MBR Master Boot Record. (2023, February 22). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/mbr-master-boot-record/
Graph Data Structure And Algorithms. (2024, January 17). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
Estefania Cassingena Navone. (2020, September 28). Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction. FreeCodeCamp.org; freeCodeCamp.org. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:~:text=Dijkstra's%20Algorithm%20finds%20the%20shortest,node%20and%20all%20other%20nodes
Binary Heap. (2014, November). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/binary-heap/
Separate Chaining Collision Handling Technique in Hashing. (2015, July 31). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing/
Hash Table Data Structure. (2023, April 24). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/hash-table-data-structure/
Quadratic Probing in Hashing. (2020, May 10). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/quadratic-probing-in-hashing/