Reflexión: Actividad integradora #5

Miguel Arriaga Velasco – A01028570

Campus Santa Fe

Programación de estructuras de datos y algoritmos fundamentales

3 de junio del 2021

1. Hashing y conjuntos

Un conjunto es una estructura de datos que apoya la implementación de la técnica de Hashing para la búsqueda de información. Al trabajar con un conjunto, lo importante es la pertenencia o no de un elemento en el conjunto.

Aplicaciones:

- Tiene aplicaciones con las operaciones típicas de conjuntos matemáticos: pertenencia, unión, diferencia, etc.
- Es útil para hacer validaciones.
- Puede funcionar como una estructura de búsqueda en tiempos extremadamente bajos.

Técnica de Hashing:

- Se realiza una función matemática a la llave del elemento que se quiere encontrar/insertar/eliminar del conjunto para localizar su posición dentro de la estructura, comúnmente una tabla.
- Lo ideal es que esa posición siempre esté disponible para él.
- Sin embargo, no es posible asegurar que la función hashing arroje un valor distinto para cada llave, a esto se le llama "colisiones".
- Es importante aplicar una buena estrategia para controlar las colisiones dentro de la tabla.

Métodos más comúnmente empleados en el diseño de funciones hash:

- Selección de dígitos: seleccionar algunos dígitos de la llave aleatoriamente.
- Residuales: utilizar como índice el residuo de dividir la llave entre el tamaño de la tabla.
- Cuadrática: elevar al cuadrado el valor de la llave y tomar los dígitos centrales.
- Folding: agrupar algunos de los dígitos de la llave y aplicarles alguna operación matemática.

En el caso de esta situación problema, se utilizará el método de residuales, ya que es sencillo de implementar y tiene un buen rendimiento en bases de datos grandes como la bitácora de IPs.

Nota: además de utilizar el método de residuales, primero la IP fue convertida a un valor "random" fabricado con una semilla igual al valor de la IP sin puntos. Esto se hizo para asegurar que no haya llaves repetidas y se llegó a esta solución después de experimentar con otras, por ejemplo, utilizar el valor numérico de la IP con ciertos parámetros de conversión. Esto último, fue bastante ineficiente ya que existían muchas llaves repetidas. Se cree que la solución a la que se llegó es muy eficiente y elegante.

Estrategias para el manejo de colisiones

- Métodos de dirección abierta: consisten en reacomodar el elemento colisionado dentro de la tabla.
    - Prueba Lineal
    - Prueba Cuadrática
    - Prueba Reasignación Aleatoria
    - Prueba Doble Hashing
- Métodos de Encadenamiento: El elemento colisionado es reacomodado en un espacio externo a la tabla.
    - Método de encadenamiento externo
    - Método de encadenamiento de colisiones

En este programa se utilizará el método de prueba cuadrática, utilizando banderas de status, ya que fue el método solicitado por la actividad.

1.1. Complejidad del hash

Para la clase “HashTable”, que es la clase encargada del almacenamiento y operaciones del hash, se decidió utilizar un arreglo de tamaño N, en donde N es el número de IPs de la bitácora, que contiene datos del tipo “HashNode”, por lo tanto, la complejidad espacial del hash es de:

$$O(n) = n$$

La complejidad temporal del método HashFunc, que es la técnica de hashing que se utilizó es constante, ya que es una simple operación de módulo:

$$O(n) = 0$$

Las operaciones para buscar e insertar un nuevo elemento (ver figura 1) se pueden analizar de manera empírica como algorítmo iterativo. El ciclo “while” que se muestra a continuación se va a continuar repitiendo mientras que el elemento no encuentre un lugar vacío en la tabla y el elemento no sea igual a la llave que se está buscando. Si observamos cuidadosamente, podemos notar que esto básicamente se traduce a que las operaciones del ciclo se van a realizar mientras existan colisiones. Ya que las operaciones dentro del ciclo son constantes, se concluye que la complejidad temporal de estos métodos es de:

$$O(n) = m$$

En donde m es el número de colisiones con las que se encuentre el valor. Sin embargo, ya que la operación hash busca minimizar el número de colisiones, la operación “Find” se aproxima a tiempo constante en la mayoría de los casos. Por lo tanto, el hash puede ser construido en tiempo casi lineal, lo que es muy bueno.

```
int HashTable::Find(int key)
{
    int pos = HashFunc(key);
    int collisions = 0;
    while (table[pos].info != Empty &&
           table[pos].element != key)
    {

        pos = pos + 2 * ++collisions -1;
        if (pos >= size)
            pos = pos - size;
    }
    //std::cout<<key<<endl;
    // std::cout<<table[pos].element<<endl;
    return pos;
}
```

Figura 1. Algorítmo para encontrar un valor en el hash.

2. Heap y Grafo

Al igual que en la actividad integradora pasada, se utilizó un Heap para poder acomodar las direcciones IP y se utilizo un grafo de Lista de Adyacencia para almacenar cada nodo y sus vecinos. Las complejidades de estas dos estructuras y sus operaciones son exactamente iguales que en la entrega pasada.

Conclusión y reflexión sobre hashing para la situación problema

La situación problema de este curso es sobre los ataques cibernéticos y la identificación de páginas infectadas, esto se hace a través de grandes bitácoras de información. Después de aprender sobre tablas hash e implementarlas, me di cuenta de que es una estructura increíblemente rápida para hacer una gran cantidad de búsquedas. A pesar de que existan algoritmos como la búsqueda binaria, que pueden encontrar elementos en tiempo logarítmico, se requiere de esta operación cada vez que se quiera buscar un elemento. Por el otro lado, las tablas hash pueden hacer búsquedas casi en tiempos constantes y cuantas tantas queramos, por lo que son mucho más eficientes cuando se esta trabajando con bases de datos así de grandes.

En un problema de esta naturaleza, un hash es una estructura de datos complementaria increíblemente poderosa y útil. Con "complementaria", me refiero a que, al combinarla con otras estructuras de datos, como un grafo, puede reducir la complejidad temporal de un programa exponencialmente, como fue el caso. Analizar patrones en los datos y combinar estructuras de datos, determinar funciones hash eficientes dependiendo del caso y seleccionar métodos de manejo de colisiones, es lo que diferencia a un ingeniero de software, de un programador.

Referencias

Presentaciones por el Dr. Eduardo Arturo Rodríguez Tello, durante la materia de Programación de estructuras de datos y algoritmos fundamentales

GeeksForGeeks (2021) "Hashing Data Structure". Recuperado desde: https://www.geeksforgeeks.org/hashing-data-structure/

GeeksForGeeks (2021) "Quadratic Probing in Hashing". Recuperado desde: https://www.geeksforgeeks.org/quadratic-probing-in-hashing/