Estructura de datos
Tablas de dispersión dinámicas
Actividad 11

Dagoberto Quevedo

10 de abril de 2020

Resumen

En esta actividad se describen las propiedades de una tabla de dispersión dinámicas (*hash table* en inglés) y se procede a la implementación computacional en Python para emular esta estructura.

1. Tablas de dispersión dinámicas

Las tablas *hash* estructuras asocian claves o llaves con valores, este tipo de almacenamiento permite una búsqueda eficiente de almacenamiento y búsqueda, tal como en el caso de los vectores, las tablas proporcionan un tiempo constante de búsqueda promedio de $\mathcal{O}(1)$, sin importar el tamaño de la tabla.

1.1. Funciones *hash*

Una función de dispersión o *hash* es una función del conjunto de claves posibles a las direcciones de memoria o posiciones de la tabla. Una buena función *hash* debe tener las siguientes propiedades: a) eficientemente computable, b) distribuir uniformemente las claves (cada posición de la tabla es igualmente probable para cada clave).

En la práctica, se emplean técnicas heurísticas para crear una función *hash* que sea eficiente. La información cualitativa sobre la distribución de las claves puede ser útil en este proceso de construcción de una función. Abordaremos dos método heurísticos usuales: *hash* por división y *hash* por multiplicación.

En el *método por división* o módulo, se asigna una clave k a unos de los *slots* de la tabla tomando el resto de la clave dividida por tamaño de la tabla ℓ. Es decir,

$$h(k) = k \mod \ell. \tag{1}$$

En el *método por multiplicación*, multiplicamos la clave k por un número real constante c en el rango $0 < c < 1$ y extraemos la parte fraccional de $k \times c$ y multiplicamos este valor por ℓ, para posteriormente tomar la función piso del resultado. Se puede representar como,

$$h(k) = \lfloor m \times (k \times c \mod 1) \rfloor \tag{2}$$

1.2. Solución de colisiones

Se produce una colisión cuando dos valores obtienen el mismo espacio, es decir, la función *hash* genera el mismo número de espacio para varios valores. Si no se toman los pasos de resolución de colisión adecuados, el elemento anterior en el *slot* será reemplazado por el nuevo elemento cada vez que ocurra la colisión.

- *Sondeo lineal*: Consisten en buscar un *slot* disponible siempre que ocurra una colisión. La búsqueda del espacio disponible inicia desde el lugar donde ocurrió la colisión y se mueve secuencialmente a través de la tabla hasta encontrar una *slot* vacío.
- *Encadenamiento*: Permite que existan múltiples elementos en el mismo *slot*. Esto puede crear una colección de valores en un mismo espacio. Cuando ocurre la colisión, el valor se almacena en el mismo *slot* usando un proceso de encadenamiento.

2. Implementación computacional

Se realiza la implementación computacional en Python que emula el funcionamiento de una estructura de una tabla *hash*, se incorpora dos métodos heurísticos para la función *hash*, división y múltiplicación y se adapta un método de solución de colisiones por encadenamiento.

2.1. Condiciones de experimentación

Se desea medir la eficiencia de las funciones *hash* así como medir el número de colisiones que se generan en la operación de inserción. La evaluación se realiza con una serie de instancias, en este caso cada instancia se define como un vector de tamaño m de valores o claves únicas a ubicar en una

tabla *hash* de tamaño n, en este caso $n = 1000$ y $m = n \times 2$, el experimento se itera $k = 5$ veces reportando para cada inserción el valor de la función *hash* calculado y si hubo o no colisión.

2.2. Resultados y conclusiones

La figura 1 muestra la frecuencia de asignación de valores en cada *slot* de la tabla, se aprecia que la función módulo distribuye de manera más eficiente las claves dentro de la estructura.

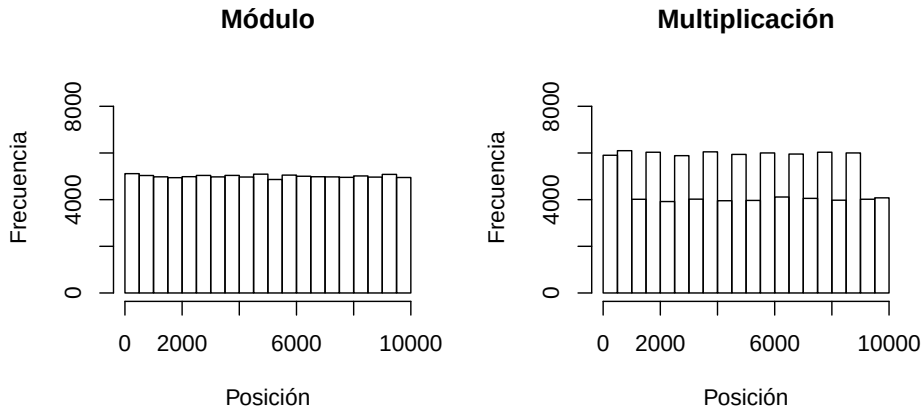

Figura 1: Resultado de la experimentación en un proceso de inserción

Respecto al número de colisiones, el método de división obtiene en promedio entre todas las iteraciones un 56 % de colisiones y el método de multiplicación cercano al 88 %, lo que refuerza el argumento de que la función módulo es eficiente y evita en menor medida el número de colisiones.

Referencias

[1] R. L. Graham, D. E. Knuth y O. Patashnik: *Concrete Mathematics: A Foundation for Computer Science*. Addison Wesley, 1994.

[2] Donald Knuth, *Sorting and searching*, The Art of Computer Programming, Addison-Wesley Professional, 1998.

[3] Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley Professional, 1993.

[4] Elisa Schaeffer, *Modelos computacionales*, Complejidad computacional de problemas y el análisis y diseño de algoritmos, notas de curso, 2020.