Universidad Simón
Bolívar

Reporte de Laboratorio de Semana 10

Análisis de implementaciones de tablas de hash

Autor:
Christopher Gómez

Profesor:
Guillermo Palma

Laboratorio de Algoritmos y Estructuras de Datos II (CI2692)

25 de julio de 2021

1. Metodología

El siguiente estudio es una comparación experimental entre dos implementaciones concretas del TAD Diccionario basadas en tablas de hash. La prueba a realizar consiste en los siguiente:

1. Se crea un arreglo **Array<Int>** que almacenará n números enteros generados aleatoriamente en el intervalo $[0..\lfloor \frac{2n}{3} \rfloor]$
2. A partir de este arreglo, se crea otro arreglo de pares $(clave, valor)$ en una estructura **Array<Pair<Int, String>>**, donde el valor asociado a cada clave se obtiene convirtiendo en **String** la clave, de tipo entero.
3. Se itera sobre cada elemento de este último arreglo de pares $(clave, valor)$ y se **busca** el elemento en la tabla de hash (inicialmente vacía), si este existe, se **elimina**, de lo contrario, se **agrega**.

El procedimiento descrito se aplicará a las dos implementaciones de tablas de hash realizadas, tomando $n =$ 1.000.000, 2.000.000, 3.000.000 y 4.000.000, y se medirá el tiempo en milisegundos que tarda cada estructura en procesar el mismo arreglo de pares. No se toma en cuenta en la medición el tiempo involucrado en la generación del arreglo a procesar.

Las implementaciones concretas del TAD Diccionario fueron hechas en el lenguaje de programación **Kotlin**, y consisten en (i) una tabla de hash basada en encadenamiento, que será referida en el informe como **HashTable** y (ii) una tabla de hash basada en una variante de direccionamiento abierto que usa el algoritmo de *hashing del cuco*, que será referida en el informe como **CuckooTable**.

Los siguientes son los detalles de la máquina y el entorno donde se ejecutaron las pruebas:

- **Sistema operativo:** Linux Mint 19.3 Tricia 32-bit.
- **Procesador:** Intel(R) Celeron(R) CPU G1610 Dual-Core @ 2.60GHz.
- **Memoria RAM:** 2,00 GB (1, 88 GB usables).
- **Compilador:** kotlinc-jvm 1.5.0 (JRE 11.0.11+9).
- **Entorno de ejecución:** OpenJDK Runtime Environment 11.0.11
- **Opciones:** -Xss10m.

2. Detalles de implementación

Ambas implementaciones son **dinámicas**, es decir, el tamaño de la(s) tabla(s) crece y se hace *rehashing* si el factor de carga es mayor o igual a un límite establecido, que para este caso es igual a 0.7 para *HashTable* y *CuckooTable*. Cada tabla comienza con un tamaño de 7 y el *rehashing* consiste en aumentar el tamaño de las mismas a $m = \lfloor \frac{3}{2}(m_0 + 16) \rfloor$, donde m_0 es el tamaño antes del *rehashing* y m el tamaño después.

Para la implementación de *HashTable* se usaron listas doblemente enlazadas con el objetivo permitir la eliminación de un nodo en tiempo constante, y la función de hash usada es $h(k) = k \bmod m$, donde m es el tamaño actual de la tabla. En cambio, para *CuckooTable* se usa esta misma función de hash como primaria y como función secundaria se usa $h_2(k) = \lfloor m(kA \bmod 1) \rfloor$, donde se toma $A = \frac{\sqrt{5}-1}{2}$, como es sugerido por Donald Knuth[1], aproximada a 0.6180339887.

En ambas implementaciones se usó aritmética de precisión simple en vez de doble para el cálculo del factor de carga, detalle que mejoró sutilmente el desempeño de las implementaciones en las pruebas sin afectar el funcionamiento interno de las estructuras.

[1] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. Introduction to algorithms, 3rd ed. MIT press, 2009.

Por otro lado, el algoritmo de *hashing del cuco* hace uso de una variable $maxIter$ que representa el número máximo de iteraciones que se harán en el ciclo para insertar una nueva clave, antes de decidir que se han hecho muchos intercambios e intentar la inserción nuevamente con una tabla de hash más grande y, por consecuencia, una función de hash distinta. El recurso teórico provisto[2] sugiere el uso de $maxIter = \lfloor 3 \log_b(n) \rfloor$, donde $b > 0$ es una constante acotada por la inversa de factor de carga y n es el número de casillas, sin embargo, la naturaleza dinámica de la tabla hacía que el cálculo del logaritmo tuviera que hacerse en cada inserción, resultando en un peor desempeño. Así, se tomó $maxIter = \frac{numElementos}{2} + 1$, elección que mostró un buen desempeño empíricamente en las pruebas realizadas; se hicieron pruebas además tomando $maxIter$ como constantes pequeñas como $1, 2, 8$ y 16 y se obtuvieron resultados destacables, sin embargo, se desecharon estas elecciones porque hacían uso excesivo de la memoria (el número de casillas disponibles era de más de 4 veces el número de claves en la tabla) y un uso mínimo de la tabla secundaria (menos del 5 % de los elementos de la tabla residían en la secundaria), por lo que se decidió escoger una variable que balancease la eficiencia en tiempo y en recursos.

3. Resultados experimentales

La Figura 1 y el Cuadro 1 muestran los resultados obtenidos de aplicar las pruebas descritas en la sección 1 a *HashTable* y *CuckooTable*.

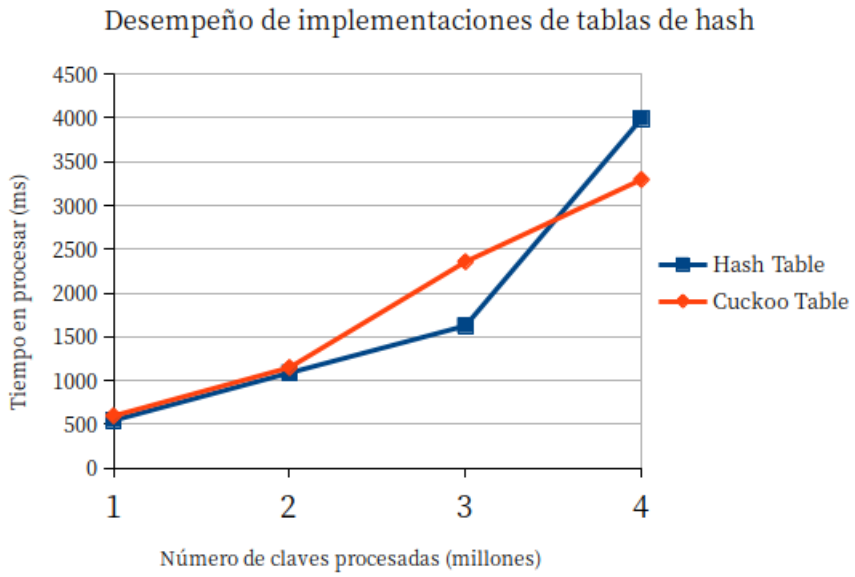

Figura 1: Tiempos de ejecución de las pruebas para las implementaciones de tablas de hash

	Tiempo (ms)	
n	HashTable	CuckooTable
1M	544	596
2M	1087	1148
3M	1625	2357
4M	3992	3296

Cuadro 1: Comparativa de tiempos entre implementaciones de las tablas de hash

Como se puede notar, el desempeño de las estructuras al procesar los arreglos menos numerosos (de 1 millón y 2 millones de elementos) es muy similar, aunque con cierta ventaja en general para *HashTable*; a lo largo

[2]Pagh, R., and Rodler, F. F. Cuckoo hashing. Journal of Algorithms 51, 2 (2004), 122–144.

de varias corridas, la diferencia absoluta entre los tiempos de *HashTable* y *CuckooTable* rondaba los 50-250 ms para 1 millón de elementos y los 100-300 ms para 2 millones, sin embargo, con mucha menor frecuencia, se lograba ver que *CuckooTable* obtenía tiempos ligeramente menores a los de *HashTable*. Por otro lado, de forma consistente, los resultados para las pruebas sobre 3 millones de elementos mostraron una ventaja para *HashTable* de entre 600 y 1100 ms, caso contrario al de las pruebas sobre 4 millones, en las que *CuckooTable* obtuvo tiempos en su mayoría de 500 a casi 2000 ms menores a los de *HashTable*. De tal manera, los datos usados para la Figura 1 y en el Cuadro 1 son tomados de una ejecución del programa de pruebas cuyos resultados representan con precisión razonable la generalidad de los resultados obtenidos a lo largo de varias ejecuciones.

Se observa en la Figura 1 que el tiempo de las ejecuciones crece de forma aproximadamente lineal para ambas estructuras, aunque con un salto notorio en las pruebas sobre $3M$ para *CuckooTable* y sobre $4M$ para *HashTable*. Debido a que en cada prueba se hacen n búsquedas, de las cuales alrededor de $n/3$ resultan exitosas y $2n/3$ infructuosas[3], derivando en eliminaciones e inserciones, respectivamente, el crecimiento aproximadamente lineal de la gráfica concuerda con lo estudiado en la teoría, dado que todas las operaciones que se hacen transcurren en $O(1)$, y son ejecutadas una cantidad de veces proporcional al tamaño n del arreglo de pares. Para *CuckooTable* se tiene que la búsqueda y eliminación se producen en tiempo constante en el peor caso, mientras que las inserción se da en un tiempo amortizado de $O(1)$; por su parte, la distribución aleatoria de las claves a procesar, y el hecho de que el factor de carga se mantenga siempre por debajo de 0.7 (es decir, es $O(1)$), nos permite concluir que para *HashTable* las búsquedas, inserciones y eliminaciones también tendrán un tiempo constante.

A pesar de que todas las operaciones de las pruebas tengan un desempeño asintótico constante, el *rehashing* tarda un tiempo lineal con respecto al número de elementos en la tabla antes de cada ampliación de tamaño, lo que contribuye a que los resultados en las pruebas muestren un comportamiento asintótico no necesariamente lineal. Se tiene que las implementaciones hechas conforme a la especificación dada sufren un total de 22 *rehashes* para las pruebas sobre 1 millón de elementos, y hasta 26 *rehashes* para los 4 millones, lo cual, dado que cada *rehash* es más costoso que el anterior, explica los saltos en los tiempos mostrados en la Figura 1.

Luego, en el Cuadro 2 se muestran los resultados obtenidos al contar la cantidad de veces que ocurre *rehashing* en el programa de pruebas para los distintos tamaños del arreglo de pares[4]. De esta forma, se puede inferir que el salto en los tiempos de *CuckooTable* se debe a que para los 3 millones de elementos se hacen 2 *rehashes* más que para 2 millones, lo cual afecta los resultados que se ven en la Figura 1, sin olvidar además que usar 2 tablas para la implementación *CuckooTable* hace que las constantes ocultas involucradas en la copia de dos arreglos, en vez de uno solo, sean ligeramente mayores.

	Cantidad de rehashes	
n	**HashTable**	**CuckooTable**
1M	23	22
2M	24	23
3M	25	25
4M	26	25

Cuadro 2: Cantidad de rehashes en las pruebas para cada implementación de tablas de hash

En el mismo orden de ideas, para 4 millones de elementos se logra observar que *CuckooTable* logra terminar de procesar el arreglo sin hacer nuevamente *rehash* desde los 3 millones, mientras que *HashTable* sufre de un último rehash. lo cual explica la consistente y notoria desventaja que obtiene con respecto a *CuckooTable* en las pruebas más numerosas.

[3]Aproximación basada un programa que contaba el número de inserciones y eliminaciones en las pruebas, distinto al usado para la comparativa.

[4]Nuevamente, se llevó esta cuenta en un programa distinto para no afectar los tiempos de la Figura 1. Los contadores arrojaron los mismos resultados en la mayoría de las ejecuciones, salvo en casos aislados donde CuckooTable llegaba a hacer 26 rehashes.

3.1. Conclusiones

Finalmente, la implementación de estas estructuras, los resultados obtenidos en las pruebas, y el análisis de los mismos permiten obtener las siguientes conclusiones:

- Cuando se trata de tablas de hash como una estructura que permite el acceso a información en tiempo constante a través de claves (más que de índices de arreglo), es necesario y sumamente importante cuidar cada detalle de las implementaciones con el fin de obtener la eficiencia que se busca como primer objetivo de la implementación misma: la elección de las funciones de hash, el tamaño de la tabla, la función de incremento de tamaño (en caso de ser dinámicas), el factor de carga aproximado que se desea mantener, etc.
- Aunque no se trata del único escenario que se puede presentar en la práctica, hacer pruebas usando la misma secuencia de claves distribuidas aleatoriamente en un intervalo es útil para chequear el correcto funcionamiento de la estructuras y entender cómo podrían desempeñarse en grandes escalas.
- No siempre se ha de priorizar solamente la eficiencia en el tiempo, sobre todo cuando se sabe que un algoritmo puede hacer un uso excesivo del espacio si opera sobre conjuntos de datos muy grandes.
- Una implementación simple de tablas de hash dinámicas con encadenamiento, con una función de hash fija y un umbral para el factor de carga pequeño, es una opción razonable cuando se puede asegurar que no existirá ningún patrón en las claves a agregar, sin embargo, el *hashing del cuco* es una alternativa que resulta competitiva y fácil de implementar, que además asegura un tiempo de búsqueda y eliminación constante en el peor de los casos.