Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-2692 - Laboratorio de Algoritmos y Estructuras de Datos II
Trimestre Abril-Julio 2021

TAD Diccionario y tablas de hash

1. Introducción

El objetivo de este laboratorio es presentar una especificación del TAD Diccionario y realizar dos implementaciones concretas basadas en tablas de hash. La primera implementación consiste en una tabla de hash basada basada en encadenamiento. La segunda implementación es una tabla de hash basada en direccionamiento abierto, en específico, la cuco hashing. Para garantizar la consistencia entre la especificación del TAD Diccionario y las implementaciones, para cada una de las implementaciones se va a definir la *relación de acoplamiento* entre el TAD Diccionario y las tablas de hash.

2. TAD Diccionario

La Figura 1 muestra una especificación del TAD Diccionario, basada en la presentada en [3] y en la clase de teoría de Algoritmos y Estructuras II. Esta especificación posee la operación `iterator` que retorna un elemento de tipo `Iterador`, que va a servir para iterar sobre los pares $(clave, valor)$ que se obtienen de la relación parcial `tabla`. EL TAD Diccionario, debe ser implementado como una interfaz de Kotlin. Todos sus métodos deben ser abstractos para que las implementaciones se encuentren en las clases concretas. El nombre de la interfaz es `Diccionario` y el archivo que la contiene debe ser llamado `Diccionario.kt`. Observe que el método `crear` corresponde al constructor de la clase que implemente la interfaz `Diccionario`.

3. Tabla de hash basada en encadenamiento

Se quiere que realice una primera implementación del TAD Diccionario por medio de la estructura de datos tabla de hash, usando el método de encadenamiento para la resolución de colisiones. Usted debe implementar una tabla de hash en donde las claves son de tipo entero, y el valor a almacenar que esta asociado a cada clave, es un elemento de tipo String. El encadenamiento debe ser llevado a cabo mediante una lista doblemente enlazada. La Figura 2 muestra un ejemplo de la tabla de hash, la cual es semejante a la que se presenta en el capítulo 11 de [1]. Como en la tabla de hash tenemos claves asociadas a valores, entonces se puede establecer la siguiente *relación de acoplamiento* entre la tabla de hash basada en encadenamiento y el TAD Diccionario:

$$\begin{aligned} conocidos &= \{claves \mid \text{Las } claves \text{ corresponden a las claves en la tabla de hash}\} \\ &\wedge \\ tabla &= \{(clave, valor) \mid \text{Pares de claves y valores asociados en la tabla de hash}\} \end{aligned}$$

Debe implementar una tabla de hash **dinámica**, es decir, el tamaño de la tabla crece si la tabla tiene un factor de carga mayor o igual a un límite establecido. Para este laboratorio, el tamaño inicial de la tabla es de siete casillas. Si el factor de carga llega a ser igual o

Especificación $\mathbb{A}$ del TAD Diccionario

Modelo de Representación

var $conocidas$: set of $T0$

var $tabla$: $T0 \nrightarrow T1$

Invariante de Representación

$conocidas = \text{dom}(tabla)$

Operaciones

proc *crear* (**out** d : Diccionario)

{ **Pre:** true }

{ **Post:** $d.conocidos = \varnothing \ \wedge \ d.tabla = \varnothing$ }

proc *agregar* (**in-out** d : Diccionario, **in** $clave$: $T0$, **in** $valor$: $T1$)

{ **Pre:** $clave \notin d.conocidas$ }

{ **Post:** $d.conocidas = d_0.conocidas \cup \{clave\} \ \wedge$
$d.tabla = d_0.tabla \cup \{(clave, valor)\}$ }

proc *eliminar* (**in-out** d : Diccionario, **in** $clave$: $T0$)

{ **Pre:** $clave \in d.conocidas$ }

{ **Post:** $d.conocidas = d_0.conocidas - \{clave\} \ \wedge$
$d.tabla = d_0.tabla - \{(clave, d_0.tabla(clave))\}$ }

proc *buscar* (**in** d : Diccionario, **in** $clave$: $T0$, **out** $valor$: $T1$)

{ **Pre:** $clave \in d.conocidas$ }

{ **Post:** $valor = d.tabla(clave)$ }

proc *existe* (**in** d : Diccionario, **in** $clave$: $T0$, **out** r : Boolean)

{ **Pre:** true }

{ **Post:** $r \equiv (clave \in d.conocidas)$ }

proc *toString* (**in** d : Diccionario, **out** s : String)

{ **Pre:** True }

{ **Post:** $s =$ String que es una representación de los pares
$(clave, valor)$ contenidos en $d.tabla$ }

proc *iterator* (**in** d : Diccionario, **out** it : Iterator)

{ **Pre:** True }

{ **Post:** $d = d_0 \ \wedge$ it posee la secuencia de pares $(clave, valor)$
que están contenidos $d.tabla$ }

proc *numElementos* (**in** d : Diccionario, **out** n : Int)

{ **Pre:** True }

{ **Post:** $n = \#conocidas$ }

Fin del TAD Diccionario

Figura 1: Especificación del TAD Diccionario

mayor a 0.7 durante las operaciones de la tabla, entonces tamaño n de la tabla se incrementa aplicando la función $incr(n) = int((n + 16) * 3/2)$, donde int es una función que redondea un número real a un número entero. Esta operación se conoce como *rehashing*.

La lista doblemente enlazada estará constituida por elementos de tipo `HashEntry`. En la

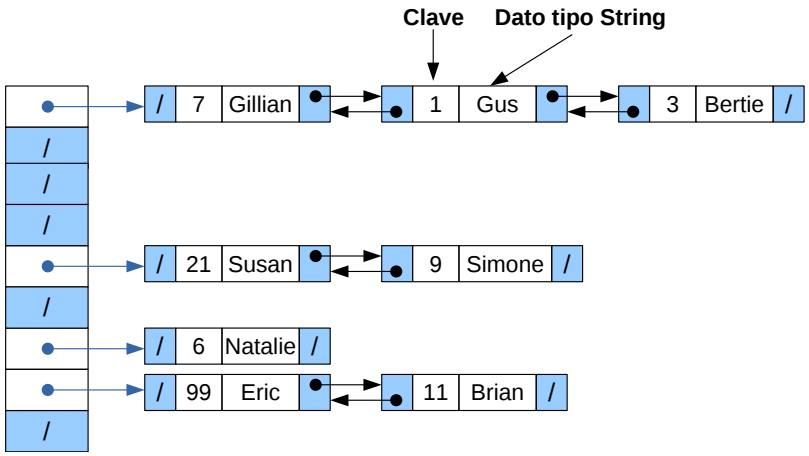

Figura 2: Ejemplo de la tabla de hash donde el encadenamiento es realizado con una lista doblemente enlazada. Las claves son elementos de tipo entero, y el valor asociado a cada clave es de tipo String.

figura 2 podemos observar que el tipo `HashEntry` debe contener al menos cuatro campos: uno para la clave, uno para el valor tipo String y dos para las referencias a otros dos elementos de tipo `HashEntry`. El tipo `HashEntry` debe ser implementado como una clase de Kotlin, cuyo constructor recibe una clave de tipo entero y un valor de tipo String.

La función de hash que debe ser usada es el método de la división que se explicó en el curso de teoría de Algoritmos y Estructuras II. De esta implementación son al menos tres los archivos que debe entregar:

`HashEntry.kt:` Contiene una clase con la implementación del tipo de datos `HashEntry`.

`DList.kt:` En este archivo debe estar implementado la lista doblemente enlazada con la clase `DList`. La lista debe contener elementos de tipo `HashEntry`.

`HashTable.kt:` En este archivo se implementa el tipo de datos tabla de hash basada en encadenamiento, en la clase `HashTable`. La clase debe hacer uso de los tipo de datos `HashEntry` y `DList`. La clase `HashTable` implementa la interfaz `Diccionario`.

4. Cuco hashing

La segunda implementación concreta del TAD Diccionario esta basada en la estructura de datos tabla de hash que usa el método de cuco hashing [2] para la resolución de colisiones. Debe implementar una tabla que usa cuco hashing en donde las claves son de tipo entero, y el valor que esta asociado a cada clave es un elemento de tipo String. La Figura 3 muestra un ejemplo de la tabla de hash.

Como en el cuco hashing tenemos claves asociadas a valores, entonces se puede establecer la siguiente *relación de acoplamiento* con el TAD Diccionario:

$$\begin{aligned} \mathit{conocidos} = \{\mathit{claves} \mid \text{Las } \mathit{claves} \text{ corresponden a las claves en la cuco hashing}\} \\ \wedge \\ \mathit{tabla} = \{(\mathit{clave}, \mathit{valor}) \mid \text{Pares de claves y valores asociados en la cuco hashing}\} \end{aligned}$$

La cuco hashing es dinámica. En este caso, se debe hacer *rehashing* cuando el factor de carga de la tabla es igual o mayor a 0.7. De debe usar la misma función indicada para la

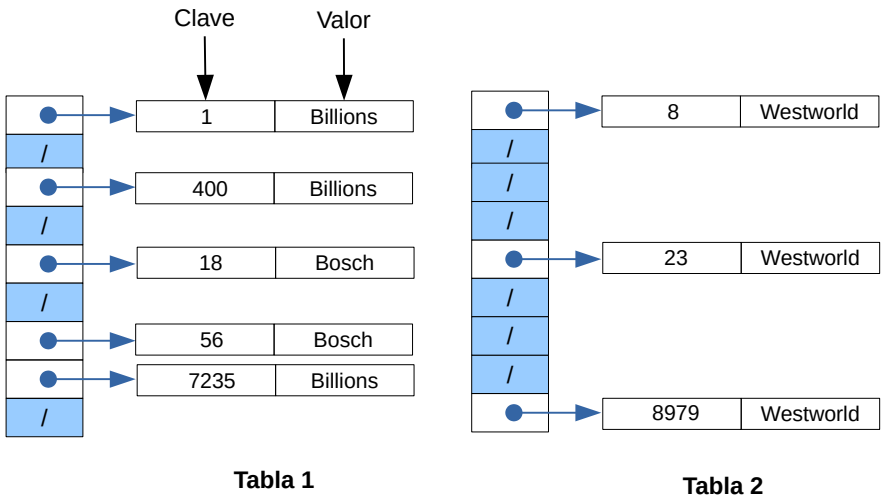

Figura 3: Ejemplo de la tabla de hash usando cuco hashing. Las claves son elementos de tipo entero, y el valor asociado a cada clave es de tipo String.

tabla de hash basada en encadenamiento, para el cálculo de tamaño de las nuevas tablas cuando se hace *rehashing*. Cuando se crea una cuco hashing, su tamaño inicial es de siete.

Cada par clave-valor en el cuco hashing, debe estar contenido en una estructura de datos llamada `CuckooEntry`. El tipo `CuckooEntry` debe poseer dos campos: el primero se llama clave, de tipo entero, y el segundo se llama valor de tipo String. El tipo `CuckooEntry` debe ser implementado como una clase de Kotlin, en el archivo `CuckooEntry.kt`. El constructor de la clase `CuckooEntry` recibe como entrada un clave, de tipo entero y un valor asociado con la clave, que es de tipo String.

La tabla de hash usando cuco hashing debe ser implementada como una clase de Kotlin, llamada `CuckooTable`. El desarrollo de tabla debe tener como base el pseudocódigo presentado en la clase del curso de teoría de Algoritmos y Estructuras de Datos II. La tabla almacenará elementos de tipo `CuckooEntry`. La `CuckooTable` debe poseer dos arreglos que contienen a los elementos de tipo `CuckooEntry`. La clase `CuckooTable` hereda de la interfaz Diccionario, y por lo tanto debe implementar todos los métodos de la interfaz.

El método de cuco hashing hace uso de dos funciones de hash h_1 y h_2. La función h_1 corresponde al método de la división que se explicó en el curso de teoría de Algoritmos y Estructuras II. Para la función h_2 se debe usar el método de la multiplicación, también dado en clase de teoría, usando como constante A el valor sugerido por Knuth [1] de $A = 0{,}6180339887$.

Su implementación de la cuco hashing debe contener, al menos, los siguientes archivos:

`CuckooEntry.kt:` Implementación de la clase `CuckooEntry`.

`CuckooTable.kt:` Implementación de la la cuco hashing en la clase `CuckooTable`.

4.1. Programa de pruebas de las tablas de hash

Se desea hacer una comparación experimental del rendimiento de la tabla de hash basada en cuco hashing y la tabla hash basada en encadenamiento. El objetivo es comparar el rendimiento de ambas tablas, bajo un conjunto de operaciones, y usando diferente cantidad de datos.

Debe realizar un programa cliente llamado `Main.kt`, el cual compara las dos tablas de hash mediante la siguiente prueba:

1. Se crea un arreglo, el cual va a almacenar n números enteros generados aleatoriamente que se encuentran en el intervalo $\left[0, \frac{2n}{3}\right]$. Este arreglo contiene las claves que van a ser agregadas en las tablas.

2. El valor asociado a cada clave se obtiene convirtiendo en String cada una de las claves, de esa manera se debe crear un arreglo de pares $(clave, valor)$ de elementos a insertar en la tabla de hash.

3. Para cada uno de los elementos del arreglo de pares $(clave, valor)$, se **busca** si el elemento existe en la tabla de hash, entonces si existe el elemento se **elimina**, de lo contrario se **agrega**.

4. Se debe medir el tiempo usado por la tabla de hash para procesar el arreglo de pares $(clave, valor)$. No se debe tomar en cuenta el tiempo usado para la creación de este arreglo.

El procedimiento descrito anteriormente debe ser aplicado a las dos tablas de hash a comparar. El cliente `Main.kt` se debe ejecutar desde un script llamado `testingHashTables.sh`, el cual funciona con la siguiente línea de comando:

```
>./testingHashTables.sh <n>
```

Donde n es el número de elementos que contiene el arreglo de pares $(clave, valor)$, que va a ser procesado por las tablas de hash. Como resultado el programa debe mostrar por la salida estándar, el tiempo de ejecución usado por cada tabla.

La entrega del programa de pruebas, junto con el resto de los códigos fuentes, debe incluir un archivo Makefile que compila todos los archivos Kotlin.

4.2. Evaluación experimental y reporte

Se debe realizar un estudio sobre el rendimiento de las tablas de hash. Se debe ejecutar el programa `testingHashTables.sh` con los siguientes valores: 1.000.000, 2.000.000, 3.000.000, y 4.000.000. El informe debe contener una tabla de resultados y su gráfica. La tabla de resultados contiene los tiempos de las dos tablas de hash para cada uno de los valores indicados anteriormente. Debe realizar entonces un gráfica tiempo de ejecución (eje y) contra el número de claves (eje x). Finalmente, debe hacer un análisis de los resultados obtenidos. El informe debe estar en formato PDF.

5. Condiciones de entrega

Los códigos del laboratorio, la declaración de autenticidad debidamente firmada, y el informe deben estar contenidos en un archivo comprimido, con formato *tar.xz*, llamado *LabSem10_X.tar.xz*, donde X es el número de carné del estudiante. La entrega del archivo *LabSem10_X.tar.xz*, debe hacerse por medio de la plataforma *Classroom* antes de las 11:50 pm del día domingo 25 de julio de 2021.

Referencias

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

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

[3] Ravelo, J. Especificación e implementación de tipos abstractos de datos. `http://ldc.usb.ve/~jravelo/docencia/algoritmos/material/tads.pdf`, 2012.

Guillermo Palma / gvpalma@usb.ve / Julio de 2021