AA2021a-06-Estructuras de datos fundamentales 2

Abdiel E. Cáceres González

10 de febrero de 2021

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.

2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE TRES

Estructuras de datos

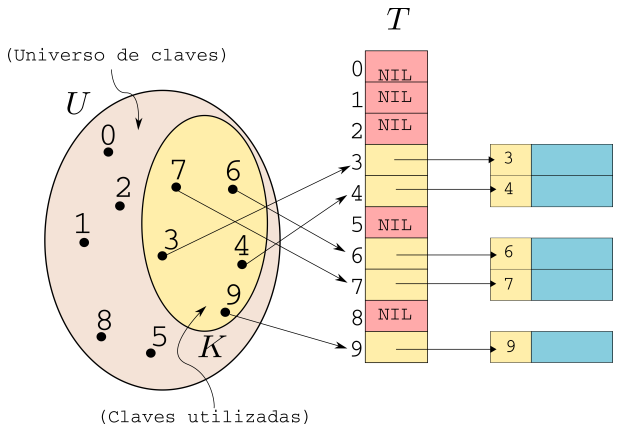
4. Tablas Hash

Muchas aplicaciones requieren un conjunto dinámico de datos que sean válidos solo para las operaciones `Insert`, `Search` y `Delete`. Por ejemplo, un compilador que traduzca un lenguaje de programación mantiene una tabla de símbolos, en la que las claves de los elementos son cadenas de caracteres arbitrarios que corresponden a identificadores en el lenguaje. Una tabla hash es una estructura de datos efectiva para implementar diccionarios. Aunque buscar un elemento en una tabla hash pudiera tomar tanto tiempo como en una lista ligada –$\boldsymbol{O}(n)$ en el peor caso– en la práctica, hacer el hash se comporta muy bien. Bajo suposiicones razonables, el tiempo promedio de la búsqueda de un elemento en una tabla hash es $\boldsymbol{O}(1)$. Podemos aprovechar el direccionamiento directo cuando tenemos un arreglo que tenga una localidad para cada posible clave. Cuando el número de claves almacenadas es relativamente mas pequeño que el número total de claves, las tablas hash se convierten en una alternativa efectiva usando direccionamiento directo sobre un arreglo, ya que una tabla hash típicamente utiliza un arreglo de tamaño proporcional al número de claves actualmente almacenadas. En lugar de utilizar la clave como un índice del arreglo directamente, el índice de arreglo se *calcula* a partir de la clave.

4.1. Tablas de direccionameinto directo

El direccionamiento directo es una técnica simple que trabaja bien cuando el universo U de claves es razonablemente pequeño. Supongamos que una aplicación funciona utilizando un conjunto dinámico de datos en donde cada elemento tiene una clave tomada del universo $U = \{0, 1, \ldots, m-1\}$, donde m no es muy grande. Asumiremos que no hay dos elementos que requieran la misma clave.

Para representar el conjunto dinámico, utilizamos un arreglo, o una **tabla de direccionamiento directo**, denotada por $T\langle 0 \ldots m-1\rangle$, en donde cada posición, o **slot**, corresponde a una clave en el universo U. En la figura siguiente se observa que el slot k apunta a un elemento en el conjunto con la clave k. Si el conjunto no tiene el elemento con la clave k, entonces $T\langle k\rangle = \texttt{NIL}$.



Las operaciones del diccionario son fáciles de implementar:

```
1  Direct_Address_Search(T,k)
2    return T[k]
```

```
1  Direct_Address_Insert(T,x)
2    T[x.key] = x
```

```
1  Direct_Address_Delete(T,x)
2    return T[x.key] = \nil
```

Cada una de estas operaciones requiere un tiempo del orden $\boldsymbol{O}(1)$.

Para algunas aplicaciones, la tabla de direccionamiento directo por si misma puede alojar los elementos en el conjunto dinámico. Esto es, en lugar de almacenar la clave del elemento y el dato satélite en un objeto externo a la tabla de direccionamiento directo, con un apuntador en el slot de la tabla al objeto, podemos almacenar el objeto en la tabla, para ahorrar espacio. Podemos utilizar una clave especial en un objeto para indicar un slot vacío. Es mas, es innecesario almacenar la clave del objeto, porque si tenemos el índice de un objeto en la tabla, tenemos su clave. Si las claves no se almacenan, debemos tener alguna manera de decir si el slot está vacío.

4.2. Tablas hash

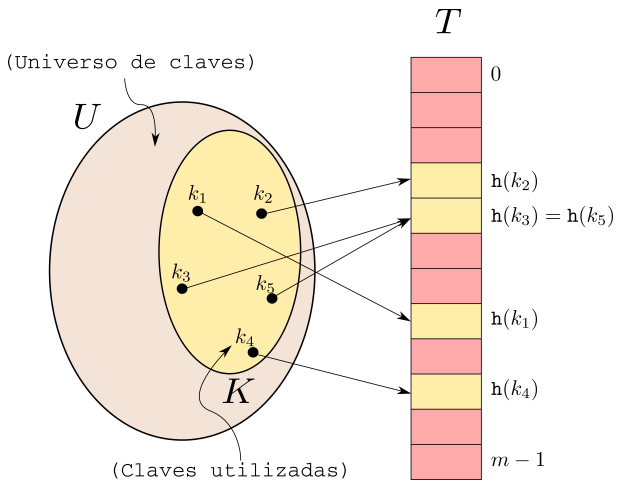
La desventaja del direccionamiento directo es obvia: si el universo U es grande, almacenar una tabla T de tamaño $|U|$ puede ser impráctico o incluso imposible, dada la memoria disponible para una computadora típica. Es mas, el conjunto K de claves utilizadas puede ser muy pequeña en comparación con U, ocasionando un gran desperdicio de espacio.

Cuando el conjunto K de claves almacenadas en un diccionario es mucho más pequeño que el universo U de todas las posibles claves, una tabla hash necesita mucho menos espacio que una tabla de direccionamiento directo. Específicamente, podemos reducir el requerimiento de espacio a $\boldsymbol{\Theta}(|K|)$ mientras mantenemos el tiempo de $\boldsymbol{O}(1)$ requerido para hacer las búsquedas en la tabla hash. Lo mejor es que esta cota es para el ctiempo en el caso promedio, mientras que en el direccionamiento directo solamente se logra en el peor caso.

Con direccionamiento directo, un elemento con clave k es almacenada en el slot k. Con hash, este elemento se almacena en el slot $\mathtt{h}(k)$; esto es, utilizamos una **función hash** $\mathtt{h}$ para calcular el slot a partir de la clave k. Aquí h mapea el universo U de claves en el conjunto de slots de la **tabla hash** $T\langle 0, \ldots, m-1\rangle$:

$$\mathtt{h} : U \to \{0, 1, \ldots, m-1\},$$

donde m es el tamaño de la tabla y típicamente $m << |U|$. Decimos que un elemento con clave k se resume [*hashes*] en el slot $\mathtt{h}(k)$; también decimos que $\mathtt{h}(k)$ es el valor resumido de la clave k. La siguiente figura miestra la idea.



La función hash reduce [resume] el rango de índices de arreglo y asi mismo el tamaño del arreglo. En lugar de un tamaño $|U|$, el arreglo tiene tama no m.

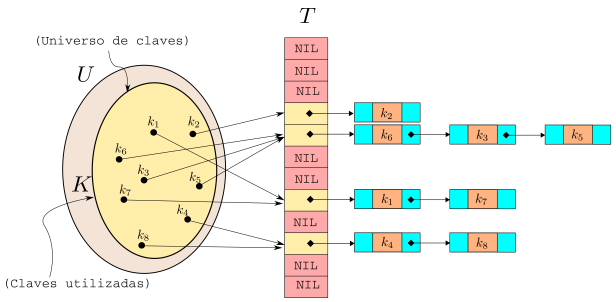
El asunto es que dos claves pueden ser resumidas en el mismo slot. Esta situación puede llamarse **colisión**. Afortunadamente, tenemos técnicas efectivas para resolver el conflico creado por las colisiones.

Claro, la solución ideal sería evitar todas las colisiones. Podríamos tratar de alcanzar este objetivo eligiendo una función hash $\mathtt{h}$ adecuada. Una idea es hacer que $\mathtt{h}$ parezca

«aleatoria», y así evitar las colisiones o al menos minimizar su cantidad. El mismo término «hash», evoca imágenes de una mezcla aleatoria que captura el espíritu de esta idea. [Claro, una función hash `h` debe ser determinista en el sentido de que al dar una clave k, se debe producir la misma salida $\texttt{h}(k)$.] Debido a que $|U| > m$, debe haber al menos dos claves que tienen el mismo valor hash; evitar todas las colisiones es imposible. Entonces, una función hash que parezca «aleatoria» puede minimizar el número de colisiones, pero aún necesitamos un método para resolver las colisiones que ocurran.

4.2.1. Resolución de conflictos por encadenamiento

En el encadenamiento, ponemos todos los elementos con el mismo hash en una misma lista ligada. El slot j contiene un apuntador a la cabeza de la lista de todos los elementos almacenados en el mismo slot j; si no hay tales elementos, el slot j contiene `NIL`.



Las operaciones de diccionario sobre una tabla hash T son fáciles de implementar cuando las colisiones son resueltas por encadenamiento:

```
1  Chained-Hash-Insert(T,x)
2    insertar x en la cabeza de la lista T[h(x.key)]

1  Chained-Hash-Search(T,k)
2    buscar un elemento con clave k en la lista T[h(k)]

1  Chained-Hash-Delete(T,x)
2    borrar x de la lista T[h(x.key)]
```

Para el caso de la inserción, el tiempo de ejecución en el peor caso es $\boldsymbol{O}(1)$. El procedimiento de inserción es rápido en parte porque se supone que el elemento x que está siendo insertado no está presente en la tabla; si fuera necesario, podemos revisar esta suposición [con un costo adicional] buscando el elemento cuya clave es $x.key$ antes de insertarlo. Para buscarlo, el tiempo de ejecución en el peor caso es proporcional a la longitud de la lista. Podemos borrar un elemento en tiempo $\boldsymbol{O}(1)$ si la lista está doblemente ligada, observa que el procedimiento `Chained-Hash-Delete(T,x)` toma como entrada un elemento x, no si clave $x.key$, de modo que no tenemos que buscar primero a x. Si la tabla hash soporta el borrado, entonces sus listas ligadas deben estar doblemente ligadas, de modo que podemos borrar un elemento rápidamente. Si las listas estuvieran simplemente ligadas, entonces para borrar un elemento x, primero tenemos que encontrar a x en la lista $T\langle\texttt{h}(x.key)\rangle$ de modo que podemos actualizar el atributo $next$ del predecesor de x. Con las listas simplemente ligadas, ambas operaciones insertar y borrar tendrían los mismos tiempos de ejección.

4.2.2. Análisis de hash con encadenamiento

Dada una tabla hash T con m slots que almacena n elementos, definimos el **factor de carga** α de T como la relación $\frac{n}{m}$, esto es, el número promedio de elementos almacenados en una cadena. Nuestro análisis será en términos de α, que puede ser menor que, igual a, o mayor que 1.

El comportamiento en el peor caso de hacer el hash con encadenamiento es terrible: todas las n claves caen en el mismo slot, creando una lista de longitud n. El peor caso para la búsqueda es así $\boldsymbol{\Theta}(n)$ mas el tiempo para calcular la función hash. Claramente, no utilizamos la tabla hash para mejorar su desempe no en el peor caso.

El desempeño en el caso promedio del hash depende de qué tan bien distribuya las claves la función hash $\mathtt{h}$ entre los m slots, en promedio. Por el momento podemos suponer que es igualmente probable que se seleccione cualquier slot, independientemente de dónde se halla almacenado cualquier otro elemento. Podemos llamar a esta suposición **hash simple y uniforme** [*simple uniform hashing*].

Para $j = 0, 1, \ldots, m-1$, digamos que la longitud de la lista $T\langle j \rangle$ por n_j. Dicho de otro modo, j toma algún valor para los slots, en cada j inicia la lista $T\langle j \rangle$ que es de tamaño n_j, por lo que

$$n = n_0 + n_1 + \cdots + \cdots + n_{m-1},$$

y el valor esperado de n_j es $E[n_j] = \alpha = \frac{n}{m}$.

Suponemos que $\boldsymbol{O}(1)$ es suficiente para calcular en valor hash $\mathtt{h}(k)$, de modo que el tiempo requerido para buscar un elemento con clave k depende linealmente de la longitud $n_{\mathtt{h}(k)}$ de la lista $T\langle \mathtt{h}(k) \rangle$. Dejando a un lado el tiempo $\boldsymbol{O}(1)$ requerido para clacular la función hash y acceder al slot $\mathtt{h}(k)$, consideraremos el número esperado de elementos examinados por el algoritmo de búsqueda, que es, el número de elementos en la lista $T\langle \mathtt{h}(k) \rangle$ que el algoritmo revisa para ver si alguno de ellos tiene clave igual a k. Vamos a considerar dos casos. En el primero, la búsqueda falla; ningún elemento en la tabla tiene la clave k. En el segundo caso, la búsqueda es exitosa encontrando un elemento con clave k.

Teorema 1. *En una tabla hash en donde las colisiones se resuelven por encadenamiento, una búsqueda fallida toma tiempo $\boldsymbol{\Theta}(1+\alpha)$ en el caso promedio, bajo la suposición de un hash simple y uniforme.*

Demostración. Bajo la suposición de un hash simple y uniforme, cualquier clave k que no se haya almacenado en la tabla es igualmente probable de que sea asignada a cualquiera de los m slots. El tiempo esperado para buscar existosamente una clave k es el tiempo esperado para buscar el final de la lista $T\langle \mathtt{h}(k) \rangle$, que tiene una longitud esperada de $E[n_{\mathtt{h}(k)}] = \alpha$. Así, el número esperado de elementos examinados en una búsqueda fallida es α, y el tiempo total requerido [incluyendo el cálculo $\mathtt{h}(k)$] es $\boldsymbol{\Theta}(1+\alpha)$ □

La situación para una búsqueda existosa es ligeramente diferente, porque no es igualmente probable que se busque en cualquier lista. En cambio, la probabilidad de que una lista sea utilizada es proporcional al número de elementos que contiene. Sin embargo, el tiempo esperado sigue siendo $\boldsymbol{\Theta}(1+\alpha)$.

Teorema 2. *En una tabla hash en donde las colisiones se resuelven por encadenamiento, una búsqueda existosa toma un tiempo promedio de $\boldsymbol{\Theta}(1+\alpha)$, bajo la suposición de un hash simple y uniforme.*

Demostración. Suponemos que todos los n elementos almacenados tienen la misma probabilidad de ser buscados. El número de elementos examinados durante una búsqueda exitosa de x es uno nas que el número de elementos que aparecen antes de x en la lista en donde está x. Como los nuevos elementos se colocan al frente de la lista, los elementos antes de x en la lista se colocaron después de que x fuera insertado. Para encontrar el valor esperado de elementos examinados, consideramos el promedio, sobre los n elementos x en la tabla, de 1 mas el número esperado de elementos agregados a la lista de x después de que x fue agregado a la lista. Digamos que x_i es el i-ésimo elemento insertado en la tabla, para $i = 1, 2, \ldots, n$, y $k_i = x_i.key$. Para las claves k_i y k_j, definimos la variable aleatoria $X_{ij} = I\{\mathtt{h}(k_i) = \mathtt{k}(k_j)\}$. Como estamos suponiendo un hash simple y uniforme, tenemos $Pr\{\mathtt{h}(k_i) = \mathtt{h}(k_j)\} = \frac{1}{m}$. Debido al lema mostrado al margen, . $E[X_{ij}] = \frac{1}{m}$.

Lema. Dado un espacio de muestras S y un evento A en el espacio de muestras S, sea $X_A = I\{A\}$, entonces $E[X_A] = Pr\{A\}$

Así el número esperado de elementos examinados en una búsqueda exitosa es:

$$\begin{aligned} E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=!}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\ &= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\ &= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\ &= 1+\frac{n-1}{2m} \\ &= 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \end{aligned}$$

Así, el número total de tiempo requerido para una búsqueda exitosa [incluyendo el cálculo de la función hash] es $\boldsymbol{\Theta}\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)=\boldsymbol{\Theta}\left(1+\alpha\right)$. □

¿Qué significa todo este análisis? Si el número de slots en la tabla hash es al menos proporcional al número de elementos en la tabla, tenemos $n=\boldsymbol{O}\left(m\right)$ y, en consecuencia, $\alpha=\frac{n}{m}=\boldsymbol{O}\left(m\right)/m=\boldsymbol{O}\left(1\right)$. Entonces, la búsqueda toma un tiempo constante en promedio. Ya que la inserción toma tiempo $\boldsymbol{O}\left(1\right)$ en el peor caso y el borrado toma también un tiempo $\boldsymbol{O}\left(1\right)$ en el peor caso cuando las listas son doblemente ligadas, podemos soportar todas las operaciones en tiempo $\boldsymbol{O}\left(1\right)$ en promedio.

4.2.3. Funciones hash

En esta parte vamos a discutir algunos aspectos relacionados con el diseño de buenas funciones hash y entonces presentar dos esquemas para su creación, uno de ellos por división y otra por multiplicación, ambas son del tipo heurísticas.

¿Qué hace que una función hash sea buena?

Una buena función hash satisface [aproximadamente] el supuesto de hacer un hash simple y uniforme: cada clave es igualmente probable de ser almacenada en cualquiera de los m slots, independientemente de dónde se han almacenado las otras claves. Desafortunadamente, no tenemos forma de verificar esta condición, ya que raramente conocemos la distribución de probabilidad de las claves. Es mas, as claves pueden no ser generadas independientemente.

De vez en cuando si conocemos la la distribución. Por ejemplo, si sabemos que las claves son números reales aleatorios k independiente y uniformemente distribuidas en el rango $0\leq k<1$, entonces la función

$$\mathtt{h}\left(k\right)=\lfloor km\rfloor$$

satisface la condición de hash simple y uniforme.

En la práctica, podemos emplear técnicas heurísticas para crear una función hash que trabaje bien. Para diseñar la función hash, puede ser útil tener infomrmación sobre la distribución de las claves. Por ejemplo, considera una tabla de símbolos de un compilador, en donde las claves son cadenas de caracteres que representan identificadores en un programa. Es posible que símbolos muy parecidos como `lst` y `lsts` ocurran en el mismo

programa. Una buena función hash debe minimizar la posibilidad de que se generen los mismos slots para esas variantes.

Un buen enfoque deriva el valor hash de manera que el valor que resulte sea independiente de cualquier patrón que pueda existir en los datos. Por ejemplo el método de la división [más adelante] calcula el slot como el resto cuando la clave se divide por un número primo específico. Este método frecuentemente da buenos resultados, suponiendo que elegimos un número primo que no esté relacionado a ningún patrón de distribución de las claves.

Interpretando claves como números naturales

La mayoría de las funciones hash asumen que el universo de claves es el conjunto $\mathbb{N} = \{0, 1, 2, \dots\}$ de los números naturales. Si las claves no son números naturales, podemos encontrar la manera de interpretarlos como números naturales. Por ejemplo, podemos interpretar una cadena de caracteres como un entero expresandola en una notación de base adecuada. Así podríamos interpretar un identificador como `lst` por la lista de números `[115, 116, 114]` porque `l` tiene el valor 115, `s` tiene el valor 116 y `t` tiene el valor 114 en la tabla ASCII; entonces hacer una representación como un entero `radix-128`, así `lst` se convierte en $\texttt{(115*128}^2\texttt{+116*128}^1\texttt{+114*128*}^0\texttt{)} = \texttt{1899122}$, dependiendo de la aplicación se debe interpretar cada clave como un número natural. En lo que sigue, supondremos que las claves son números naturales.

4.2.4. El método de la división

En el método de la división para crear funciones hash, mapeamos una clave k en uno de los m slots tomando el resto de k dividido por m. Esto es, la función hash es

$$\texttt{h}\,(k) = k \bmod m$$

Por ejemplo, si la tabla hash tiene $m = 12$ slots y la clave es $k = 100$, entonces $\texttt{h}\,(k) \rightarrowtail 4$. Como la función requiere una sola división, el cálculo del slot es muy rápido.

Cuando utilizamos el método de la división, usualmente evitamos ciertos valores de m. Por ejemplo m podría no ser una potencia de 2, porque si $m = 2^p$, entonces $\texttt{h}\,(k)$ es exactamente los p bits de menor orden de k. A menos que sepamos que todos los p-bits de menor orden son igualmente probables de ocurrir es mejor que diseñemos una función hash que dependa de todos los bits de la clave.

Un número primo que no esté cerca de una potencia de 2, usualmente es una buena opción para m. Por ejemplo, supongamos que deseamos asignar una tabla hash, con colisiones resueltas por encadenamiento, que sirva para contener aproximadamente $n = 2000$ cadenas de caracteres, donde un caracter tiene 8 bits. No nos importa examinar un promedio de 3 elementos en una búsqueda fallida, por lo que asignamos una tabla hash de tamaño $m = 701$. Podramos elegir $m = 701$ porque es un primo cerca de $\frac{2000}{3}$ pero no cerca de ninguna potencia de 2. Tratando cada clave k como un número entero, nuestra función hash sería $\texttt{h}\,(k) = k \bmod 701$.

4.2.5. El método de la multiplicación

El método de la multiplicación para crear funciones hash opera en dos pasos:

1. Multiplicamos la clave k por una constante A en el rango $0 < A < 1$ y extraemos la parte fraccionaria de kA.

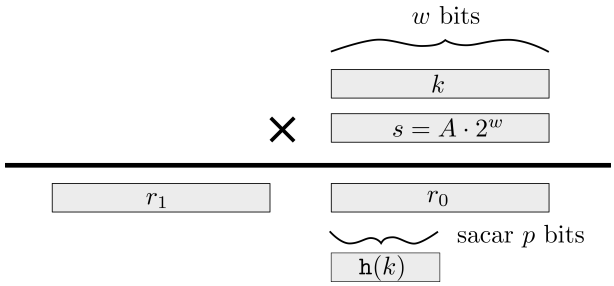
2. Multiplicamos este valor por m y fomamos el suelo del resultado.

En resumen, la función hash es

$$\texttt{k}\,(h) = \lfloor m(kA \bmod 1) \rfloor,$$

donde «$m(kA \bmod 1)$» significa la parte fraccional de kA, que es, $kA - \lfloor kA \rfloor$.

Una ventaja del método de multiplicación es que el valor de m no es crítico. Típicamente escojemos m como una potencia de 3 [$m = 2^p$ para algún entero p], porque podemos implementar más facilmente la función en la mayoría de las computadoras como sigue. Supongamos que el tamaño de la palabra de la máquina es w bits y que k cabe en una sola palabra. Restringimos A a una fracción de la forma $\frac{s}{2^w}$, donde s es un entero en el rango $0 < s < 2^w$.



La representación de w bits de la clave k es multiplicada por el valor $s = A \cdot 2^w$ también de w bits. El resultado se expresa como una palabra de $2w$ bits. Los p bits de mayor orden de la mitad de menor orden del resultado forman el valor deseado $\mathtt{h}(k)$.

Observa en la figura, primero multiplicamos k por $s = A \cdot 2^w$, ambos números representados como parabras de w bits. El resultado es una palabra de $2w$ bits $r_1 2^w + r_0$, donde r_1 es la palabra de mayor orden del producto y r_0 es la palabra de menor orden del producto. El valor hash deseado es una palabra de p bits, tomados de la parte más significativa de r_0.

Aunque este método funciona con cualquier valor de la constante A, funciona mejor con algunos valores que con otros. La elección óptima depende de las características de los datos que se están almacenando. Una sugerencia que trabaja razonadamente bien y que fue hecha por Knuth es que

$$A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots.$$

Ejemplo 1

Supongamos que $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ y $w = 32$. De acuerdo con la sugerencia de Knuth, elegimos A como una fracción de la forma $\frac{s}{2^{32}}$ que es cercana a $\frac{\sqrt{5}-1}{2}$, asi que $A = \frac{2654435769}{2^{32}}$. Entonces $k \cdot s = 327706022297554 = (76300 \cdot 2^{32}) + 17612864$, $r_1 = 76300$ y $r_0 = 17612864$. Los 14 bits mas significativos de r_0 dan el valor $\mathtt{h}(k) = 67$.

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Demuestra qué pas cuando insertamos las claves 5,28, 19, 15, 20, 33, 12, 17, 10 en una tabla hash con colisiones resueltas por encadenamiento. Digamos que la tabla tiene 9 slots y que la función hash es $\texttt{h}(k) = k \bmod 9$.

2. Escribe un programa en `Python` que se llame `hashing.py` que nos servirá para comparar los dos métodos de crear funciones hash vistas en esta lección. Debes crear dos funciones hash, ambas funciones toman como entrada un número k. La primera llamada `hdiv` genera un hash en base al método de la división, y la otra se debe llamar `hmul` que está basada en el método de la multiplicación. Debes hacer otro programa que almacene 100 números enteros aleatorios en el rango de 0 a 4999, en una tabla hash de 15 slots. Para hacer la comparación, tomaremos M el conjunto de números generados y almacenaremos el mismo conjunto de números utilizando ambas funciones. La resolución de conflictos es mediante una lista simplemente enlazada. Debes responder ¿qué función hash ofrece una uniformidad mayor?

   a) Crea la clase `Node`, con los atributos `key` y `next`. Sin métodos.

   b) Crea la clase `List`, con el atributo `head` y los métodos:

   + `show`() `:None`, para mostrar el contenido de la lista simplemente ligada.
   + `empty`() `:Boolean`, para determinar si la lista está vacía o no.
   + `Linsert`(k) `:None`, para agregar un elemento a la lista.

   c) Crea la clase `THash`, para crear dos tablas hash con los atributos: `m` pra la cantidad de slots; `h` para determinar si se ocupa la función hash `hdiv` o `hmul`; `T` para generar una lsita de `m` slots; y `A` como una constante `A = 0.6180339887` que se ocupa en caso de utilizar `hmul`. Los métodos de la clase son:

   + `hdiv`(k) `:Integer`, función hash basada en división, para generar un slot.
   + `hmul`(k) `:Integer`, función hash basada en división, para generar un slot.
   + `insert`(k) `:None`, para insertar una clave `k` en la tabla hash. La clase debe tomar la clave, generar el slot que corresponde e insertarlo en la lista de ese slot.
   + `show`() `:None`, para mostrar el contenido de la tabla hash. Se debe mostrar contenido de las listas slot por slot.

d) Crea dos tablas hash, llamadas `H1` y `H2` ambas de tamaño 15. La tabla hash `H1` debe utilizar la función hash basada en la división y la tabla hash `H2` debe utilizar la función hash basada en la muliplicación.

e) Crea un arreglo llamado `U` con 100 números aleatorios en el rango de 0 a 5000.

f) Inserta los números de `U` en ambas tablas hash.

g) Muestra el contenido de ambas tablas.

```
In []: H1 = THash(15,'div')
  ...: H2 = THash(15,'mul')
  ...: U = [random.randint(0,5000) for i in range(0,100)]
  ...: for i in U:
  ...:  H1.insert(i)
  ...:  H2.insert(i)
  ...: H1.show()
  ...: H2.show()
============================
-------[ SLOT 0 ]------
1555 4139 4071 1199 610 2631 4770 280 225
-------[ SLOT 1 ]------
3919 4207 3733 4385 2589
-------[ SLOT 2 ]------
188 1539 1518 429 3945 2340
-------[ SLOT 3 ]------
379 290 4005 2120 2065 4704
-------[ SLOT 4 ]------
28 439 371 3743 1913
-------[ SLOT 5 ]------
829 1405 2782 4358 1871 4934 342 664 185
-------[ SLOT 6 ]------
546 4748 3494 1554 3253 4761 601 3151 1999 135 25
-------[ SLOT 7 ]------
415 4986 4109 3143 161 1614 4363
-------[ SLOT 8 ]------
4567 640 695 4677 420 4855 3724 2593
-------[ SLOT 9 ]------
3242 2687 1365 1645 3339
-------[ SLOT 10 ]------
4729 3975 2878 616 4119 3132 4208 3200 370
-------[ SLOT 11 ]------
2226 1239
-------[ SLOT 12 ]------
2532 723 1744 469
-------[ SLOT 13 ]------
1736 1490 4019 4676 359 3655 2668 4663 1503 3566 1270
-------[ SLOT 14 ]------
1906 2262 1406
============================
============================
-------[ SLOT 0 ]------
4005 3975 3945 1365 420 2340 135 4770 225
-------[ SLOT 1 ]------
616 1906 601 3151
-------[ SLOT 2 ]------
3242 2687
-------[ SLOT 3 ]------
723 1518 1503
-------[ SLOT 4 ]------
379 829 4729 3919 1744 469 439 1999 3724 664
-------[ SLOT 5 ]------
290 2120 1490 695 3200 4385 185
-------[ SLOT 6 ]------
546 4986 2226 4071 4761 2631
-------[ SLOT 7 ]------
4567 2782 4207
-------[ SLOT 8 ]------
4748 188 3143 4208 4358 3743 1913
-------[ SLOT 9 ]------
4119 1539 429 1554 1239 4704 1614 2589 3339
-------[ SLOT 10 ]------
1555 1405 415 2065 640 3655 4855 610 370 280 25 1270 1645
-------[ SLOT 11 ]------
1736 161 4676 371 1871 3566 1406
-------[ SLOT 12 ]------
2532 3132 4677 2262 342
-------[ SLOT 13 ]------
28 2878 3253 3733 2668 4663 4363 2593
-------[ SLOT 14 ]------
4139 4109 3494 4019 359 1199 4934
============================
In []:
```