Tablas Hash

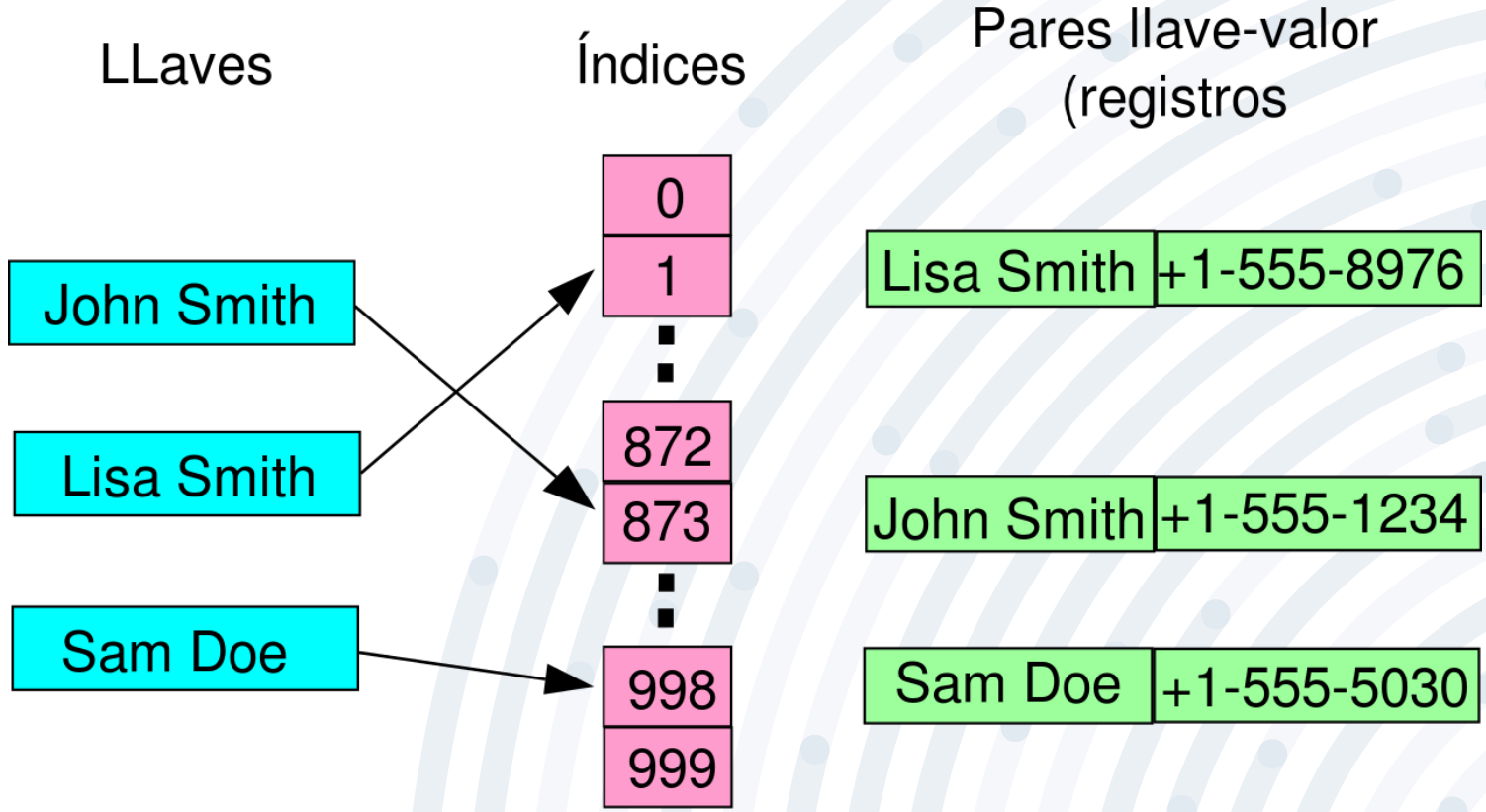
Tablas Hash

- Una **tabla hash**, matriz asociativa, hashing, mapa hash, tabla de dispersión o tabla fragmentada es una estructura de datos que asocia *llaves* o *claves* con *valores*.

- La operación principal que soporta de manera eficiente es la *búsqueda*.

- Por ejemplo, permite el acceso a los elementos (teléfono y dirección) almacenados a partir de una clave generada (usando el nombre o número de cuenta).

- Funciona transformando la clave con una **función hash** en un *hash*, un número que identifica la posición donde la tabla hash localiza el valor deseado.


LLaves
Índices
Pares llave-valor
(registros
John Smith
Lisa Smith
Sam Doe
0
1
872
873
998
999
Lisa Smith
+1-555-8976
John Smith
+1-555-1234
Sam Doe
+1-555-5030

Inserción

- La forma de implementar en función esta operación es pidiendo la llave y el valor, para con estos poder hacer la inserción del dato.
    1. Para almacenar un elemento en la **tabla hash** se ha de convertir su clave a un número. Esto se consigue aplicando la función hash a la clave del elemento.
    2. El resultado de la función hash ha de *mapearse* al espacio de direcciones del vector que se emplea como tabla, lo cual se consigue con la función módulo. Tras este paso se obtiene un índice válido para la tabla.

3. El elemento se almacena en la posición de la tabla obtenido en el paso anterior.

- Si en la posición de la tabla ya había otro elemento, se ha producido una colisión.
- Este problema se puede solucionar asociando una lista a cada posición de la tabla, aplicando otra función o buscando el siguiente elemento libre.
- Estas posibilidades han de considerarse a la hora de recuperar los datos.

Búsqueda

- La forma de implementar en función esta operación es pidiendo la llave y con esta devolver el valor.

1. Para recuperar los datos, es necesario únicamente conocer la clave del elemento, a la cual se le aplica la función hash.

2. El valor obtenido se mapea al espacio de direcciones de la tabla.

3. Si el elemento existente en la posición indicada en el paso anterior tiene la misma clave que la empleada en la búsqueda, entonces es el deseado. Si la clave es distinta, se ha de buscar el elemento según la técnica empleada para resolver el problema de las colisiones al almacenar el elemento.

- Para usar una *tabla hash* se necesita:
  - Una estructura de acceso directo (normalmente un *arreglo*).
  - Una estructura de datos con una clave.
  - Una función *hash* cuyo dominio sea el espacio de claves y su imagen (o rango) los números naturales.

- Cuando el número de claves realmente almacenadas es pequeño en relación con el número total de posibles claves, las tablas hash se convierten en una alternativa eficaz para direccionar directamente un arreglo, ya que una tabla hash normalmente usa un arreglo de tamaño proporcional al número de claves realmente almacenadas.

- En lugar de usar directamente la clave como un índice de arreglo, el índice del arreglo se calcula a partir de la clave.

Tabla de direcciones directas

- El direccionamiento directo es una técnica simple que funciona bien cuando el universo ***U*** de llaves es razonablemente pequeño.

- Supongamos que una aplicación necesita un conjunto dinámico en el que cada elemento tiene una clave extraída del universo **U = {0, 1, …., m – 1}**, donde **m** no es demasiado grande.

- Supondremos que no hay dos elementos con la misma clave.

- Para representar el conjunto dinámico, usamos un arreglo, o tabla de direcciones directas, denotada por **T[0 ... m–1]**, en el que cada posición, corresponde a una clave en el universo **U**.

- La figura 1 ilustra este enfoque; el espacio **k** apunta a un elemento del conjunto con la llave **k**.

- Si el conjunto no contiene ningún elemento con clave **k**, entonces **T[k] = NULO**.

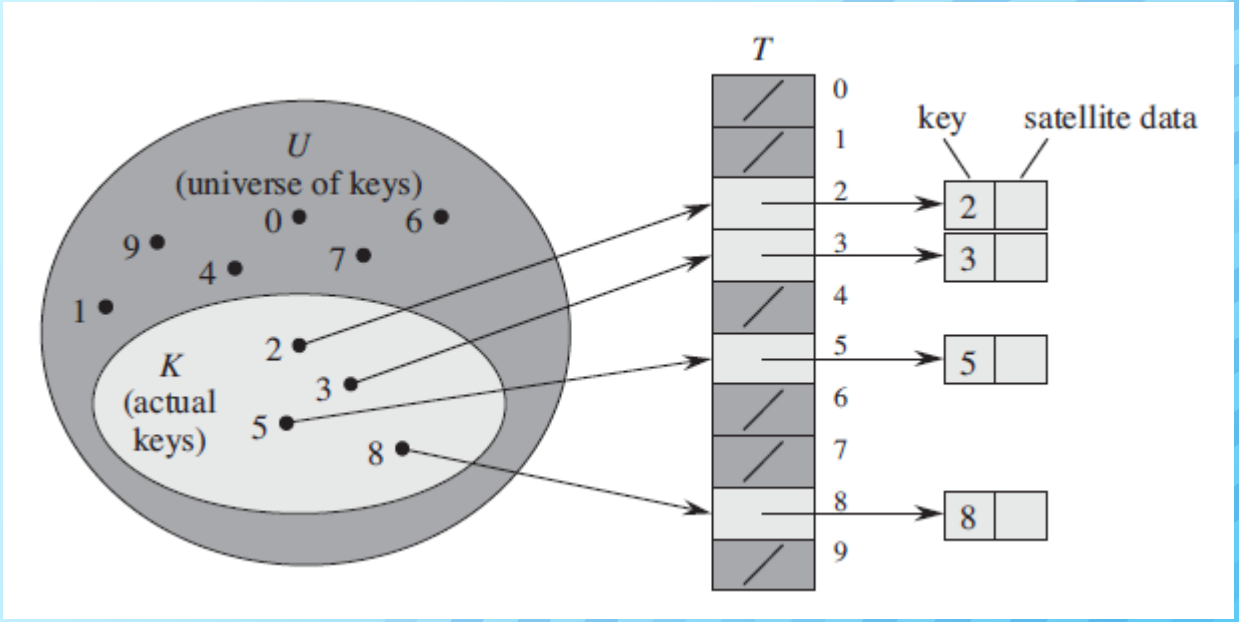


Figura 1. Cómo implementar un conjunto dinámico mediante una tabla de direcciones directas **T**.

- Cada clave del universo **U = {0, 1, ..., 9}** corresponde a un índice de la tabla.
- El conjunto **K = {2, 3, 5, 8}** de llaves actuales determina los espacios en la tabla que contienen punteros a elementos.
- Los otros espacios, más sombreados, contienen *NULO*.

- Las operaciones son triviales de implementar:

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

- Cada una de estas operaciones toma solo un tiempo de O(1).

Tablas Hash

- La desventaja del direccionamiento directo es obvia: si el universo **U** es grande, el almacenamiento de una tabla **T** de tamaño **|U|** puede ser impráctico, o incluso imposible, dada la memoria disponible en una computadora típica.

- Además, el conjunto **K** de claves realmente almacenadas puede ser tan pequeño en relación con **U** que la mayor parte del espacio asignado para **T** sería desperdiciada.

- Con direccionamiento directo, un elemento con la clave **k** se almacena en el espacio **k**.

- Con hashing, este elemento se almacena en el espacio ***h(k)*** es decir, usamos una **función hash h** para calcular la dirección de la llave **k**.

- Aquí, **h** mapea el universo **U** de claves en los índices de una tabla hash **T[0, …., m−1]**.

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

- donde el tamaño **m** de la tabla hash es típicamente mucho menor que **|U|**.

- Decimos que un elemento con clave **k** mapea en el espacio **h(k)**.
- También decimos que **h(k)** es el valor hash de la clave **k**.
- La figura 2 ilustra la idea básica. La función hash reduce el rango de índices en el arreglo y, por lo tanto, el tamaño del arreglo.
- En lugar de un tamaño de **|U|**, el arreglo puede tener un tamaño **m**.

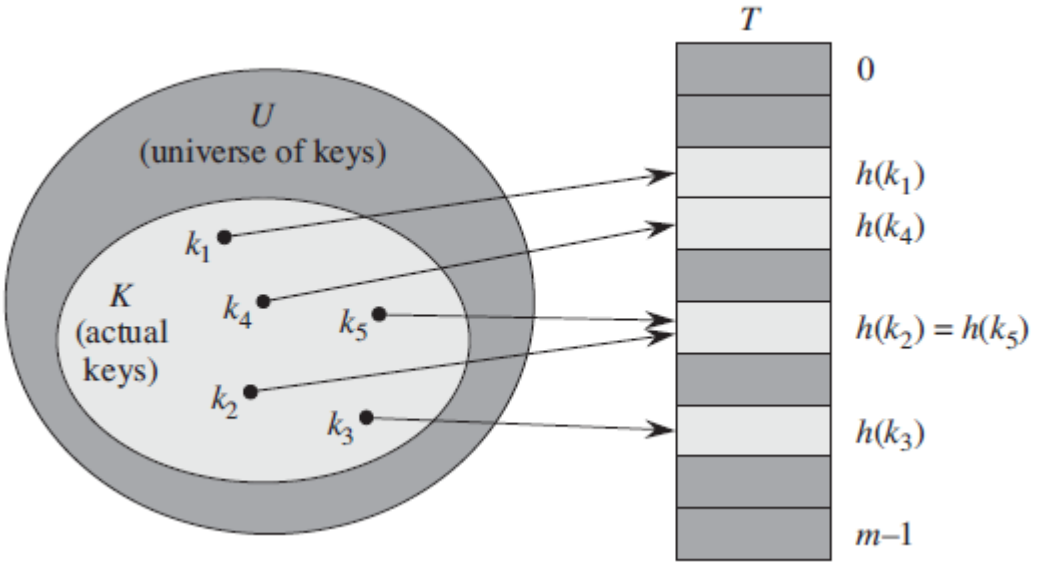


Figura 2. Uso de una función hash **h** para asignar claves en índices de la tabla hash.
Note que las claves k_2 y k_5 mapean en el mismo espacio y colisionan.

- Hay un problema: dos claves pueden introducirse en el mismo espacio. Llamamos a esta situación una colisión.
- Afortunadamente, contamos con técnicas efectivas para resolver el conflicto creado por colisiones.
- Por supuesto, la solución ideal sería evitar las colisiones por completo.
- Podríamos tratar de lograr este objetivo eligiendo una función hash adecuada **h**.
- Una idea es hacer que **h** parezca "aleatoria", evitando así colisiones o al menos minimizando su número.

- Una función hash **h** debe ser determinista en el sentido de que una entrada **k** dada siempre debe producir la misma salida **h(k).**

- Debido a que **|U| > m**, puede suceder que al menos dos claves tengan el mismo valor hash; por lo tanto, evitar las colisiones por completo es imposible.

- Si la función hash "aleatoria" esta bien diseñada, entonces se puede minimizar el número de colisiones, pero aún así, todavía se necesita un método para resolver las colisiones que puedan ocurrir.

Resolución de colisiones por encadenamiento

- En el encadenamiento, colocamos todos los elementos que tienen el mismo hash en una misma lista enlazada, como muestra la Figura 3.

- La ranura **j** contiene un puntero al primer elemento de la lista con todos los elementos almacenados con hash en la posición **j**.

- Si no hay tales elementos, el espacio **j** contiene NULO.

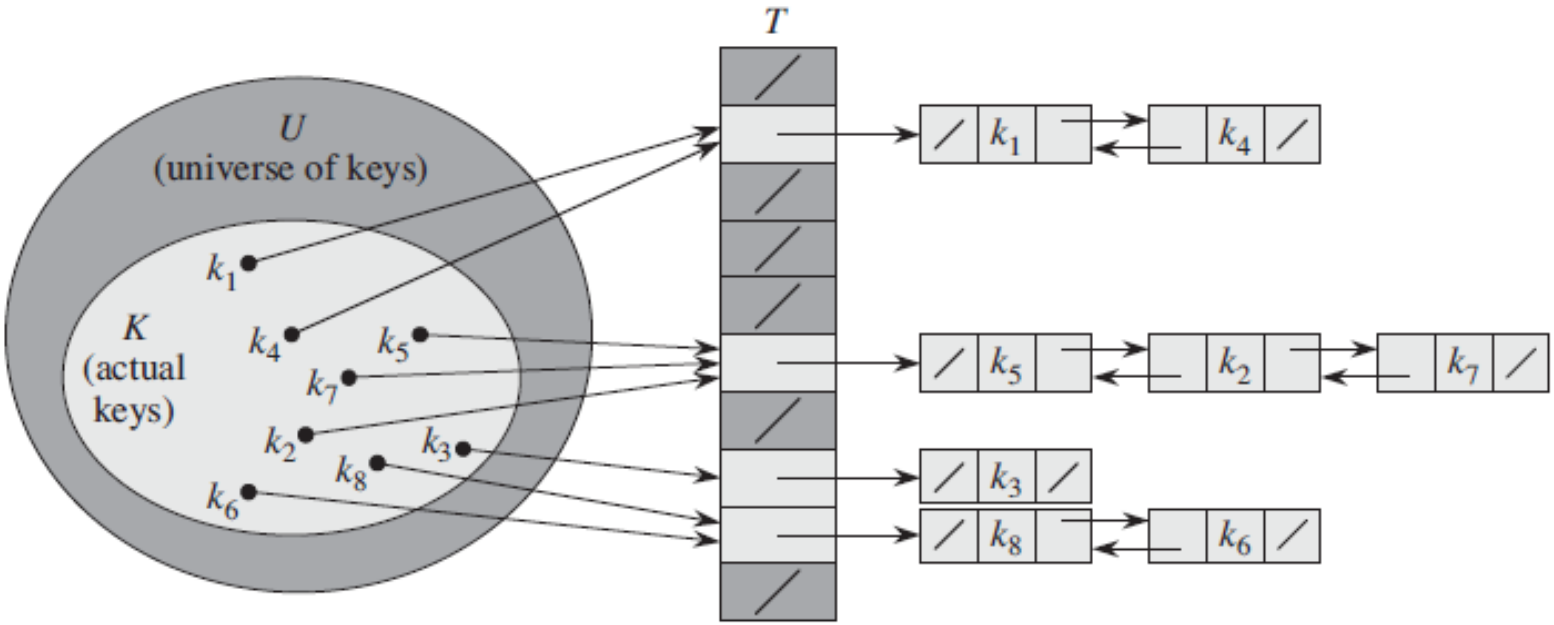


Figura 3. Resolución de colisiones por encadenamiento.

Cada ranura de tabla hash **T[j]** contiene una lista enlazada de todas las claves cuyo valor hash es **j**. Por ejemplo, $h(k_1) = h(k_4)$ y $h(k_5) = h(k_7) = h(k_2)$.

La lista enlazada puede estar simple o doblemente enlazada.

- Las operaciones en una tabla hash T son fáciles de implementar cuando las colisiones se resuelven por encadenamiento:

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

Funciones Hash

- Las **Funciones Hash** son funciones que, utilizando un algoritmo matemático, transforman un conjunto de datos en un código numérico con una longitud fija.

- Da igual la cantidad de datos que se utilice en la clave, el código resultante tendrá siempre el mismo número de caracteres.

¿Qué es lo que hace buena a una función hash?

- Una buena función hash cumple con que cada clave tiene la misma probabilidad de mapearse en cualquiera de los espacios **m**.

- Desafortunadamente, no tenemos forma de comprobar esta condición, ya que rara vez conocemos la distribución de probabilidad de las claves.

- Por ejemplo, considere la tabla de símbolos de un compilador, cuyas claves son cadenas de caracteres que representan identificadores en un programa.

- Cercanamente los símbolos relacionados, como **pt** y **pts**, suelen aparecer en el mismo programa.

- Una buena función hash minimizaría la posibilidad de que dichos símbolos se transfieran al mismo espacio.

Interpretar claves como números naturales

- La mayoría de las funciones hash asumen que el universo de claves es el conjunto N = {0, 1, 2, ……} de números naturales.
- Por lo tanto, si las claves no son números naturales, debemos de encontrar una manera de interpretarlos como números naturales.
- Por ejemplo, podemos interpretar una cadena de caracteres como un número entero expresado en notación de una base adecuada.

- Por tanto, podríamos interpretar el identificador pt como el par de enteros decimales *(112, 116)*, ya que *p = 112* y *t = 116* en el juego de caracteres ASCII; luego, expresado como un número entero radix-128, pt se convierte en:

$$pt = (112)(128)^1 + 116(128)^0 = 14452.$$

$$pts = (112).(128)^2 + (116)(128)^1 + (115)(128)^0$$
$$pts = 1849971$$

$$1849971 binario = 0001\ 1100\ 0011\ 1010\ 0111\ 0011$$

- Podemos idear algún método de este tipo para interpretar cada clave como un número natural.

- De aquí en adelante, asumimos que las claves son números naturales.

El método de la división

- En el método de división para crear funciones hash, asignamos una clave **k** a una de **m** espacios tomando el resto de **k** dividido por **m**. Es decir, la función hash es:

$$h(k) = k \bmod m$$

- Por ejemplo, si la tabla hash tiene un tamaño **m = 12** y la clave es **k = 100**, entonces ***h(k) = 4***.

- Dado que solo requiere una sola operación de división, el hash por el método de la división es bastante rápido.

- Cuando usamos el método de división, generalmente evitamos ciertos valores de **m**.

- Por ejemplo, **m** no debería ser una potencia de **2**, ya que si $m = 2^P$, entonces **h(k)** es solo los **p** bits menos significativos de **k**.

- Todos los patrones p-bit menos significativos son igualmente probables, lo mejor es diseñar la función hash para que dependa de todos los bits de la clave.

p = 2
m = 2^2 = 4
k = 100
100_2 = 01100100
h(k) = 100%4 = 0
0_2 = 00000000

p = 3
m = 2^3 = 8
k = 100
100_2 = 01100100
h(k) = 100%8 = 4
4_2 = 00000100

p = 3
m = 2^3 = 8
k = 50
50_2 = 00110010
h(k) = 50%8 = 2
2_2 = 00000010

p = 4
m = 2^4 = 16
k = 125
125_2 = 01111101
h(k) = 125%16 = 13
13_2 = 00001101

- Un número primo no demasiado cercano a una potencia exacta de **2** suele ser una buena opción para **m**.

- Por ejemplo, supongamos que deseamos asignar una tabla hash, con colisiones resueltas por encadenamiento, para contener aproximadamente **n = 2000** cadenas de caracteres, donde un carácter tiene **8** bits.

- No nos importa examinar un promedio de **3** elementos en una búsqueda sin éxito, y así que asignamos una tabla hash de tamaño **m = 701**.

- Podríamos elegir **m = 701** porque es un primo cerca de **2000/3** pero no cerca de ninguna potencia de **2**.

- Tratando cada clave **k** como un entero, nuestra función hash sería:

$$h(k) = k \bmod 701$$

- Ejemplos en donde la función **h(k)** nos da el mismo valor para diferentes valores **k** y por lo tanto se produciría una colisión.

h(k) = k mod m	h(k) = k mod m
m = 12 k = 100	m = 12 k = 64
h(100) = 100 mod 12	h(64) = 64 mod 12
100/12 = 8 12 * 8 = 96 100 - 96 = 4	64/12 = 5 12 * 5 = 60 64 - 60 = 4
h(100) = 4	h(64) = 4

El método de la multiplicación

- El método de multiplicación para crear funciones hash opera en dos pasos.
- Primero, multiplicamos la clave **k** por una constante **A** en el rango **0 < A < 1** y extraemos la parte fraccionaria de **kA**.
- Luego, multiplicamos este valor por **m** y hacemos un redondeo de piso del resultado.
- En resumen, la función hash es:

$$h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$$

- donde "kA mod 1" significa la parte fraccionaria de kA, es decir:

$$\text{“}kA \bmod 1\text{”} = kA - \lfloor kA \rfloor$$

- Una ventaja del método de multiplicación es que el valor de **m** no es crítico.

- Por lo general, lo elegimos para que sea una potencia de **2**.

- **$m = 2^P$** para algún número entero **p**.

- Supongamos que el tamaño de palabra de la máquina es **w** bits y que **k** cabe en una sola palabra.

- Nosotros podemos restringir **A** para que sea una fracción de la forma $s = 2^w$, donde **s** es un número entero en el rango:

$$0<s<2^w$$

- Primero multiplicamos **k** por el entero w-bit $s=A\cdot 2^w$.

- El resultado es un valor de **2w** bits $r_1 2^w + r_0$, donde r_1 es la palabra de orden superior del producto y r_0 es la palabra de orden inferior del producto.

- El valor hash de **p-bit** deseado consta de los **p** bits más significativos de r_0.

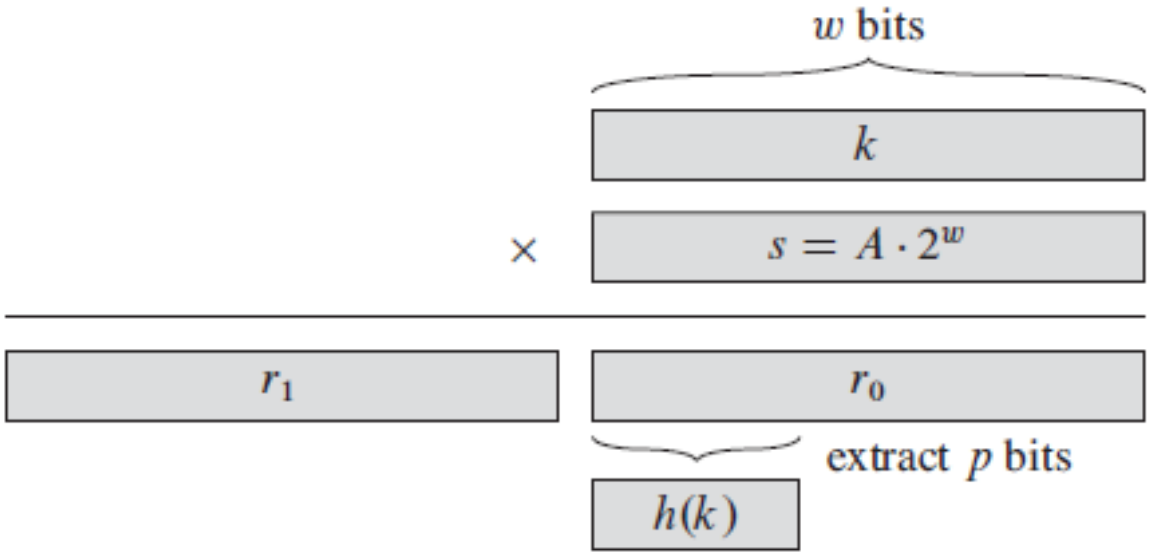


Figura 4. El método de multiplicación de hash.

La representación w-bit de la clave **k** se multiplica por el valor de w-bit **$s = A \cdot 2^w$**.

Los **p** bits de orden más alto de la mitad inferior de **w** bits del producto forman el valor hash deseado **h(k)**.

- Aunque este método funciona con cualquier valor de la constante **A**, funciona mejor con unos valores que con otros.

- La elección óptima depende de las características de los datos que se procesan.

- Knuth sugiere que si:

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

es probable que funcione razonablemente bien.

- Como ejemplo, suponga que tenemos:

  k = 123456
  p = 14
  m = 2^{14} = 16384
  w = 32

- Adaptando la sugerencia de Knuth, elegimos **A** como la fracción de la forma $\mathbf{s/2^{32}}$ que está más cerca de $\mathbf{(\sqrt{5}-1)/2}$

- De modo que: A = 2654435769 / 2^{32} = 0.61803

- Luego: k·s = 327706022297664 = (76300·2^{32}) + 17612864

- y así: r_1 = 76300 r_0 = 17612864

- Los **14** bits más significativos de $\mathbf{r_0}$ producen el valor **h(k) = 67**.

Clase universal de funciones Hash

- Comenzamos eligiendo un número primo **p** lo suficientemente grande para que cada posible clave **k** esté en el rango de **0 a p – 1**.
- Sea $Z_p = \{0, 1, \ldots, p - 1\}$
- Sea $Z^*_p = \{1, 2, \ldots, p - 1\}$
- Asumimos que el tamaño del universo de claves es mayor que el número de espacios en la tabla hash, entonces se tiene que **p > m**.

- Ahora definimos la función hash ***hab*** para cualquier:

  a ∈ Z*p **b** ∈ Zp

- usando una transformación lineal seguida de reducciones del módulo **p** y luego del módulo **m**:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

- Por ejemplo, con **p = 17** y **m = 6**, tenemos $h_{3,4}(8) = 5$.

$$\begin{aligned} H_{3,4}(8) &= ((3)(8) + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 \end{aligned}$$

Direccionamiento abierto

- En el direccionamiento abierto, todos los elementos ocupan la propia tabla hash.
- Es decir, cada entrada en la tabla contiene un elemento del conjunto dinámico o NULO.
- Al buscar un elemento, examinamos sistemáticamente los espacios de la tabla hasta que encontramos el elemento deseado o hemos comprobado que el elemento no está en la tabla.

- No se almacenan listas ni elementos fuera de la tabla, a diferencia del encadenamiento.

- Así, en direccionamiento abierto, la tabla hash puede "llenarse" para que no se puedan realizar más inserciones.

- Por supuesto, podríamos almacenar las listas enlazadas por encadenamiento dentro de la tabla hash, en los espacios de tabla hash que de otro modo no se usarían, pero la ventaja del direccionamiento abierto es que evita los punteros por completo.

- En lugar de seguir los punteros, calculamos la secuencia de espacios a examinar.

- La memoria extra liberada por no almacenar punteros proporciona a la tabla hash un mayor número de espacios para la misma cantidad de memoria, lo que potencialmente produce menos colisiones y una recuperación más rápida.

- Hay tres técnicas de uso común para calcular las secuencias de direccionamiento abierto:
  - Linear probing
  - Quadratic probing
  - Double hashing

- Linear probing

  Given an ordinary hash function $h' : U \to \{0, 1, \ldots, m-1\}$, which we refer to as an ***auxiliary hash function***, the method of ***linear probing*** uses the hash function

  $$h(k, i) = (h'(k) + i) \bmod m$$

  for $i = 0, 1, \ldots, m-1$.

- Quadratic probing

  Quadratic probing uses a hash function of the form

  $$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m ,$$

  where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \ldots, m-1$.

- Double hashing

  $$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

  where both h_1 and h_2 are auxiliary hash functions.