Algoritmos y Estructura de Datos

Unidad 2 – Semana 9

Docentes:

- Bancayán Reátegui, Mónica Lourdes
- Calderon Vilca, Hugo David
- Canaval Sanchez, Luis Martin
- Cueva Chavez, Walter

Logro de sesión

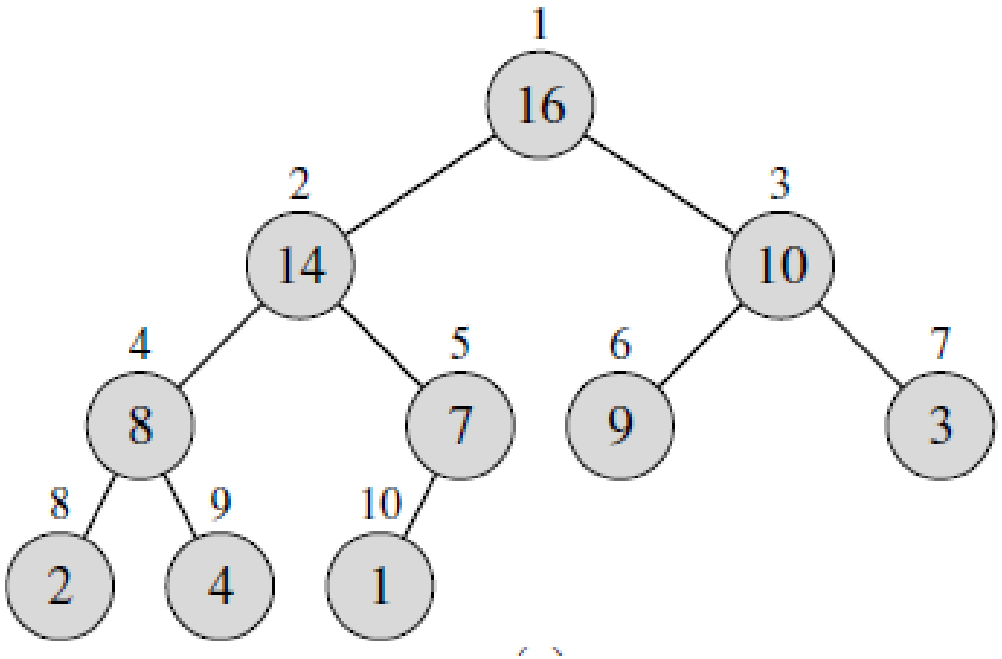
Al finalizar la sesión, el estudiante implementa estructuras Heap y Tablas de Hash

Sesión 6 : Estructuras de datos

Contenido:

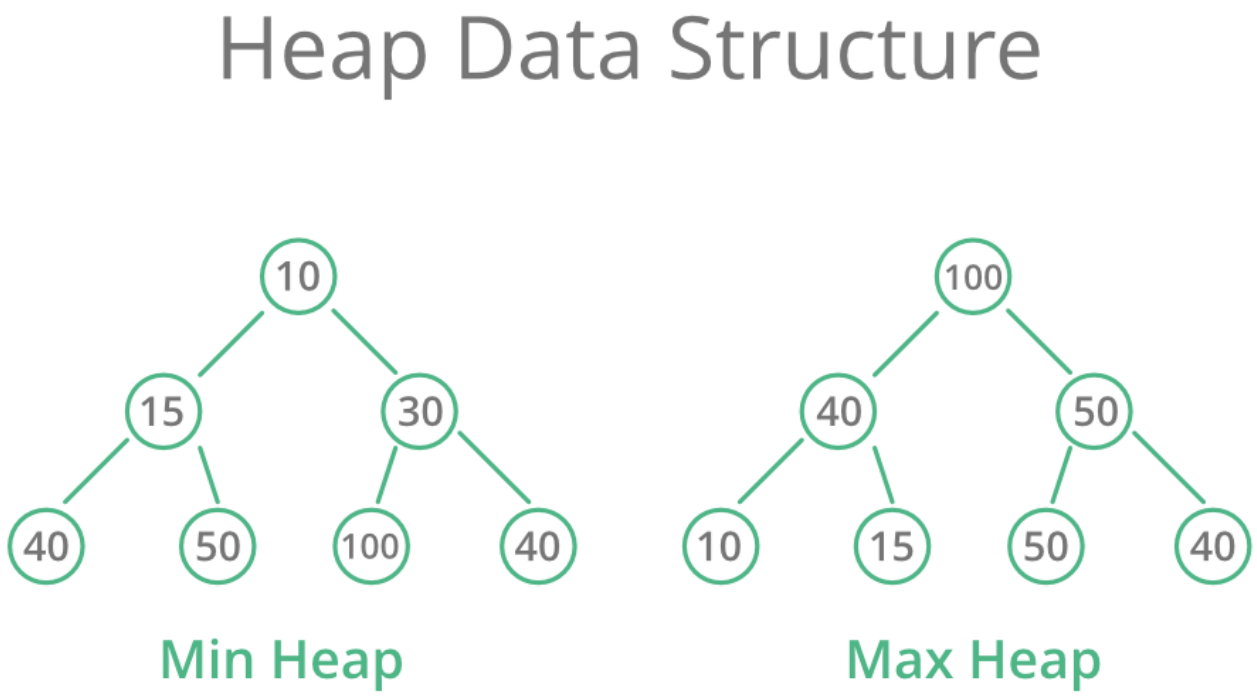
- Heap
- Tablas de Hash

HEAP

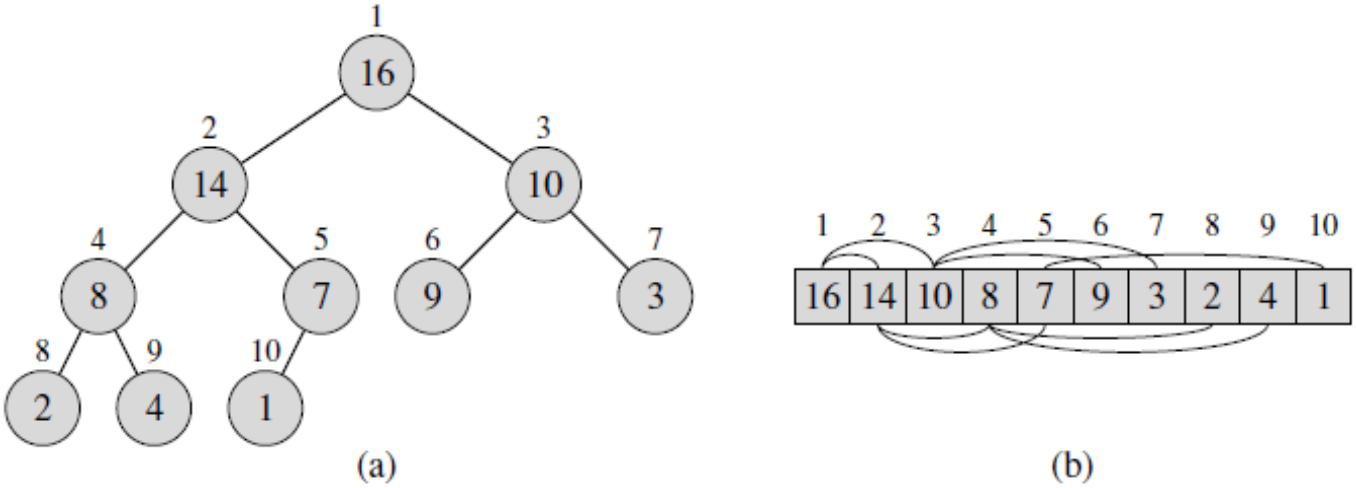


Heap

- Es una estructura de datos especial basado en árbol, en que el árbol es un árbol binario completo.
- Generalmente puede ser de dos tipos:
  - **Max-Heap:** El nodo raíz debe ser el mayor que todos sus nodos hijos.
  - **Min-Heap:** El nodo raíz debe ser el menor que todos sus nodos hijos



Heap



- Representación
- (a) Un árbol binario
- (b) Su arreglo

Heap

- Aplicaciones:
  - **Heap sort:** Heap Sort usa Binary Heap para ordenar una matriz en tiempo O (nLogn).
  - **Priority Queue:** Las colas de prioridad se pueden implementar de manera eficiente utilizando Binary Heap porque admite las operaciones insert (), delete () y extractmax (), disminución de KeyKey () en el tiempo O (logn). Binomoial Heap y Fibonacci Heap son variaciones de Binary Heap. Estas variaciones realizan la unión también de manera eficiente.
  - **Graph Algorithms:** Las colas de prioridad se usan especialmente en algoritmos de gráficos como la ruta más corta de Dijkstra y el árbol de expansión mínima de Prim.

Heap

- El elemento raíz del árbol es A[1] dado el índice i de un nodo, los índices de su padre PARENT(i), hijo izquierdo LEFT(i) y el hijo de la derecha RIGHT(i) se obtienen de la siguiente forma:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

Heap

– En max-heap, la propiedad max-heap es la de cada nodo i aparte de la raíz,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

es decir, el valor de un nodo es como máximo el valor de su padre.

– En min-heap, se organiza en el camino opuesto; la propiedad min-heap es que para cada nodo que no sea el raíz,

$$A[\text{PARENT}(i)] \leq A[i] .$$

el elemento más pequeño en un montón mínimo está en la raíz.

Heap

- Operaciones básicas
  - MAX-HEAPIFY: que tiene un tiempo asintótico de O(log n), es la clave para el mantenimiento de la propiedad de max-heap.
  - BUILD-MAX-HEAP: que tiene un tiempo asintótico de O(1), produce un máximo heap de una entrada de arreglo desordenado.
  - HEAPSORT: que tiene un tiempo asintótico de O(n log n), ordena un arreglo en su lugar.
  - MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, y HEAP-MAXIMUM, tienen un tiempo asintótico de O(log n), permiten usar la estructura de datos heap como una cola de prioridad.

Heap

- Implementación:

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

Heap

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

Heap

- Cola de prioridad máxima

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

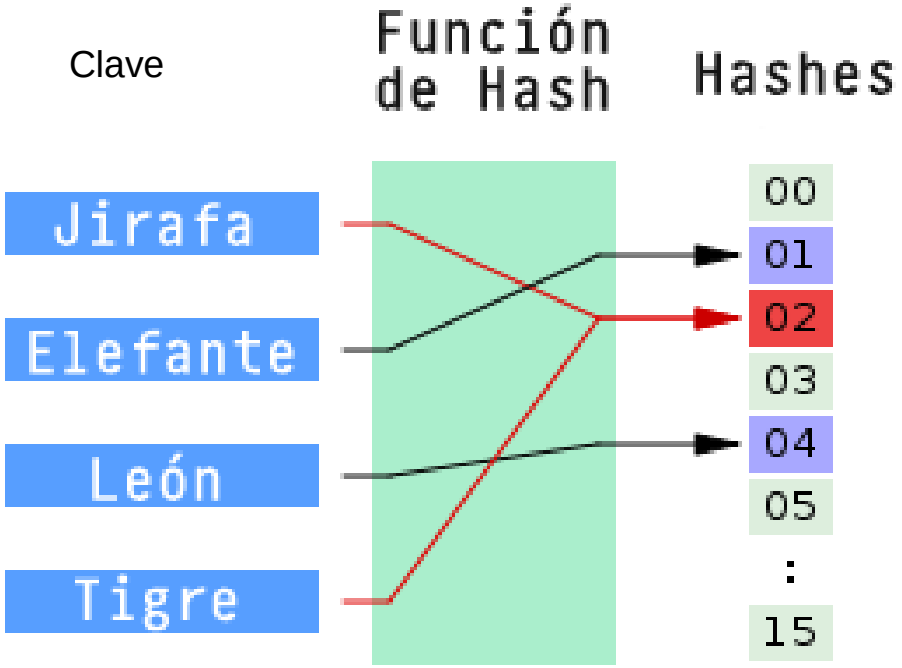
```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

Ejercicio

- Escriba el pseudocódigo para los procedimientos de HEAP-MINIMUN, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, y MIN-HEAP-INSERT que implementa una cola de prioridad mínima.

TABLAS DE HASH



Tablas de Hash

- Usa un arreglo menor
- Convierte las claves en uso en índices de la tabla a través de una función.
- Esta función de mapeo es la *función hash*.
- La tabla así organizada es la *tabla hash*.

h(k) = i

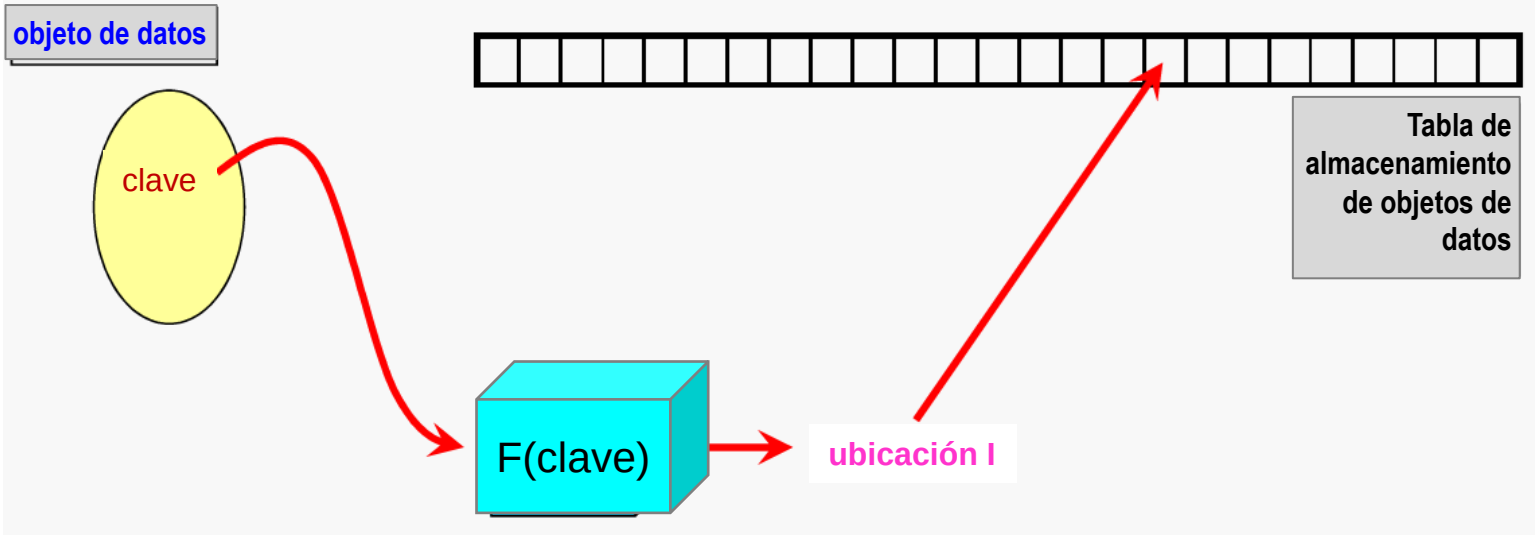
h: Función de hash

k: Clave

i: Dirección o índice en la tabla

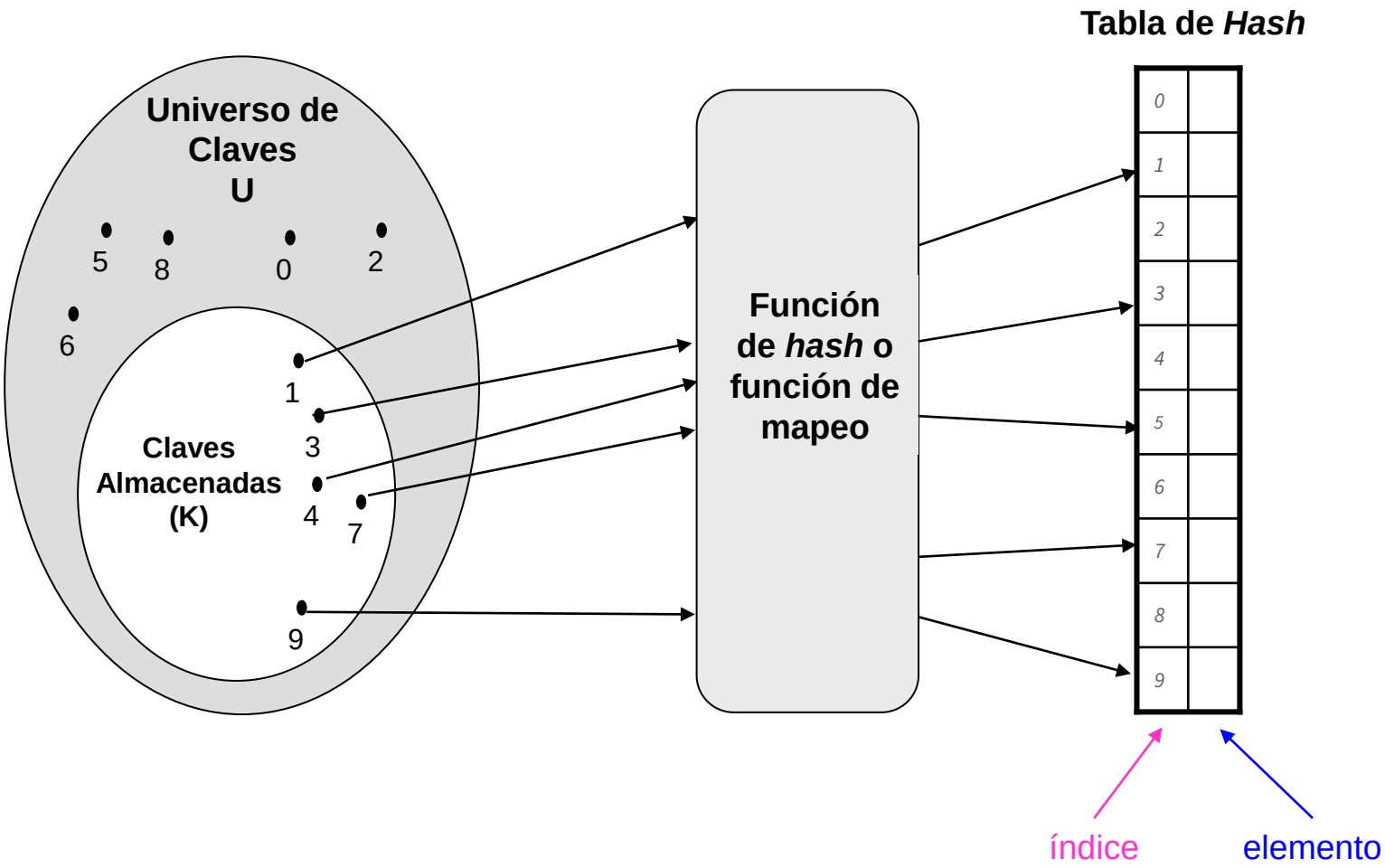
Tablas de Hash

- Es una estructura de datos que vincula claves con índices.



- Apoya con eficiente las operaciones de búsqueda.
- Permite el acceso a los elementos almacenados mediante una clave generada.
- Se implementa sobre vectores de una dimensión.

Tablas de Hash



Tablas de Hash

- Es un **diccionario de datos** que permite realizar operaciones de manera rápida.
- Se espera que el tiempo de búsqueda sea constante O(1).
- Utiliza una transformación de una clave para poder accesar al dato, mediante un índice.
- Es una estructura de datos que permite como mínimo: insertar, buscar y eliminar elementos.

Tablas de Hash

- Operaciones

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else i ← i + 1
7    until i = m
8  error "hash table overflow"
```

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4            then return j
5          i ← i + 1
6    until T[j] = NIL or i = m
7  return NIL
```

Desventajas

- No se puede recorrer el contenido de una tabla hash.

- Como la posición de un clave se calcula de forma matemática, los elementos no pueden almacenarse ordenados.

- Si permitimos elementos duplicados se producen “colisiones”.
- En algunos casos dos claves pueden tener la misma posición en el arreglo presentándose una “colisión”.

- No es sencillo volver del índice a la clave sin la función.

Proceso de obtención de tabla de hash

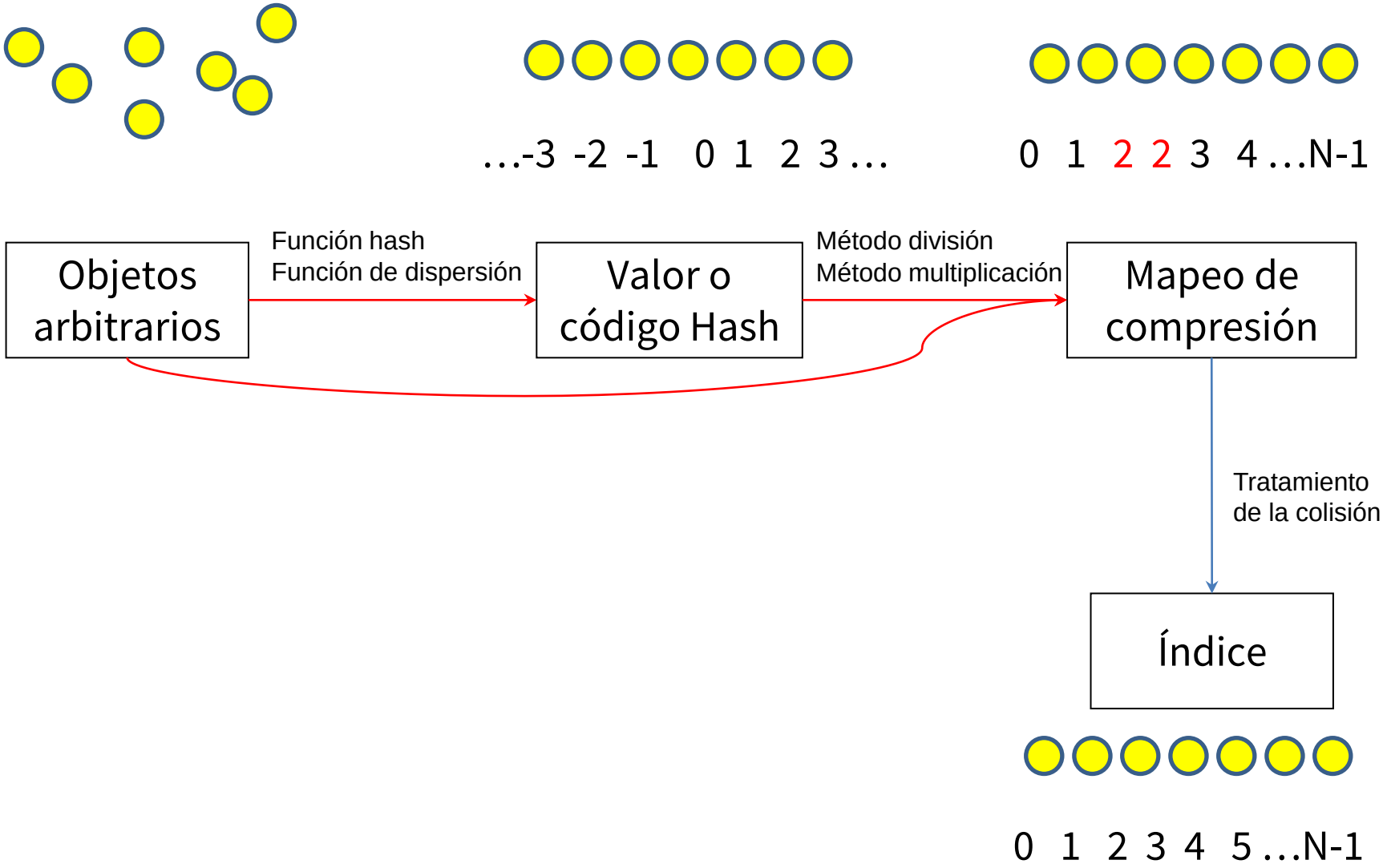


TABLA DE HASH

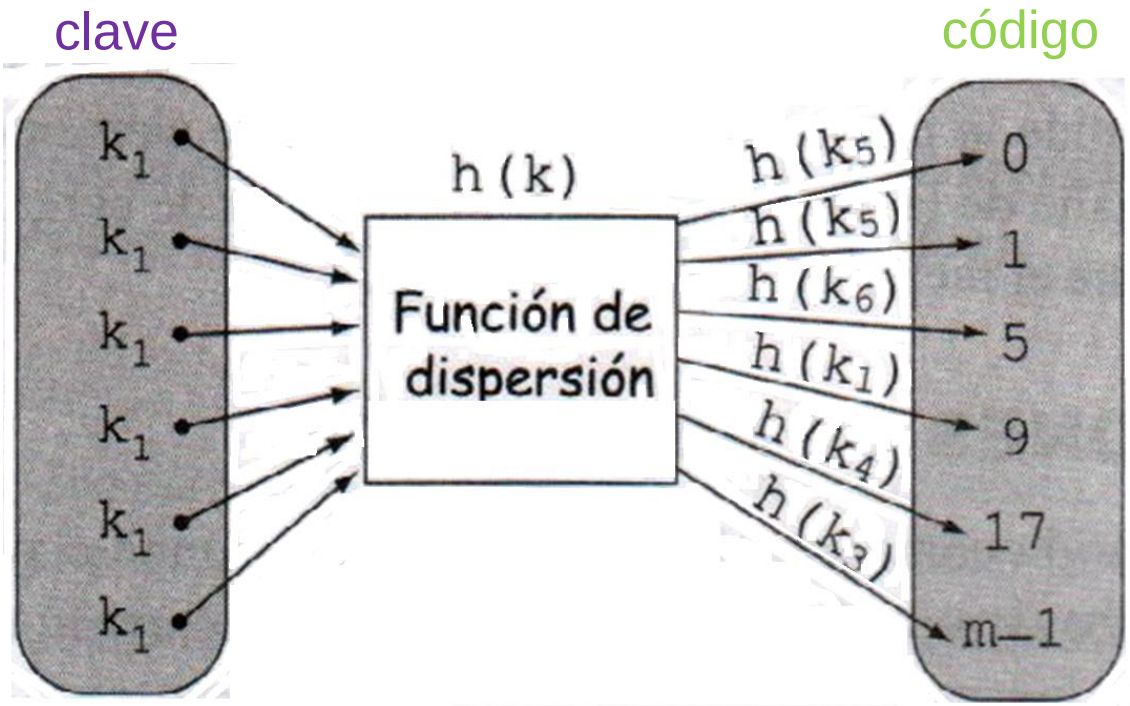
Funciones de dispersión

Mapas de compresión

Gestión de las colisiones

Función Hash (Función de Dispersión)

- Es una función que convierte la entrada (clave) en un valor entero (código hash o valor hash), con el fin de indexar la tabla en la entrada que se desea almacenar



Función Hash (Función de Dispersión)

- También se le denomina función de hash.
- Toma una clave y le asigna un valor entero.
- El entero asignado a una clave se llama código hash o valor hash.
- Este valor no necesita encontrarse en el intervalo [0 N-1], y hasta puede ser negativo, pero es preferible que el conjunto de códigos hash eviten las colisiones lo más que se pueda.

Suma de componentes

Suma del código ascii.

- Se reemplaza cada letra por su valor ascii y luego se suman.

- hash1("alfredo") = 97 + 108 + 102 + 114 + 101 + 100 + 111 = 733

- hash1(“cosa”)
- hash1(“saco”)
- hash1(“soca”)

Ejercicio

- Usando el método de la suma del código ascii, calcule el código hash de las siguientes claves.

- zapatito
- patito
- zapato

Suma de componentes

Problemas con la función

- Colisión 1: Palabras idénticas.
- Colisión 2: Palabras formadas por las mismas letras pero en distinto orden.
- Colisión 3 : Palabras distintas cuyas letras dan la misma suma.
- La suma da números muy pequeños.

Acumulación polinómica

Clave como números naturales

- Si una clave es una cadena de caracteres se puede interpretarse como un número entero en una determinada base, considerando cada caracter como un dígito.

- Si el alfabeto es {a, b, . . . , z}
- podemos asignarle a cada letra un valor del 0 al 26

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	

- hash2(“slp”) $= 19\ 27^2 + 12\ 27^1 + 16\ 27^0$
- hash2(“sdlo”) $= 19\ 27^3 + 4\ 27^2 + 12\ 27^1 + 15\ 27^0$

Ejercicio

- Usando el método clave como números naturales, calcule el código hash de las siguientes claves.

- zapatito
- patito
- zapato

Implementación

- Función Hash simple

```
1   int hash( const string & key, int tableSize )
2   {
3       int hashVal = 0;
4
5       for( char ch : key )
6           hashVal += ch;
7
8       return hashVal % tableSize;
9   }
```

Implementación

- Una buena Función Hash

```
1   /**
2    * A hash routine for string objects.
3    */
4   unsigned int hash( const string & key, int tableSize )
5   {
6       unsigned int hashVal = 0;
7
8       for( char ch : key )
9           hashVal = 37 * hashVal + ch;
10
11      return hashVal % tableSize;
12  }
```

TABLA DE HASH

Funciones de dispersión

Mapas de compresión

Gestión de las colisiones

Mapas de compresión

- El código hash para una clave no será adecuado, para usarse de inmediato en un arreglo.

- Porque el intervalo de códigos hash posibles para claves será mayor.

- Por lo tanto se requiere mapear, o relacionar los valores con los índices en el intervalo [0, N-1].

Método de división o aritmética modular

- Elegir el número de elementos que deseamos almacenar en nuestra tabla Hash, como N.
- Elegir este número como el primo(N) mayor a nuestro numero de elementos a almacenar
- N = mayor(primo(N))
- Se evita que $N = 2^k$
- De tal forma que el índice = h(k) = |k| mod N

Ejercicio

- Usando la función hash2
- Aplique el procedimiento de los mapas de compresión.
- h(Palabra) = hash2(Palabra) mod Tamaño
- h("alfredo") = $(7.27^7 + 6.27^6 + 5.27^5 + 4.27^4 + 3.27^3 + 2.27^2 + 1.27^1)$ mod 7

Ejercicio

- Analice el comportamiento de: v mod N, para v <= N.
- Verifique su comportamiento cuando N es primo.

Aritmética modular

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	1	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	0	1	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	1	2	1	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	0	0	2	1	0	6	6	6	6	6	6	6	6	6	6	6	6	6
7	1	1	3	2	1	0	7	7	7	7	7	7	7	7	7	7	7	7
8	0	2	0	3	2	1	0	8	8	8	8	8	8	8	8	8	8	8
9	1	0	1	4	3	2	1	0	9	9	9	9	9	9	9	9	9	9
10	0	1	2	0	4	3	2	1	0	10	10	10	10	10	10	10	10	10
11	1	2	3	1	5	4	3	2	1	0	11	11	11	11	11	11	11	11
12	0	0	0	2	0	5	4	3	2	1	0	12	12	12	12	12	12	12
13	1	1	1	3	1	6	5	4	3	2	1	0	13	13	13	13	13	13
14	0	2	2	4	2	0	6	5	4	3	2	1	0	14	14	14	14	14
15	1	0	3	0	3	1	7	6	5	4	3	2	1	0	15	15	15	15
16	0	1	0	1	4	2	0	7	6	5	4	3	2	1	0	16	16	16
17	1	2	1	2	5	3	1	8	7	6	5	4	3	2	1	0	17	17
18	0	0	2	3	0	4	2	0	8	7	6	5	4	3	2	1	0	18
19	1	1	3	4	1	5	3	1	9	8	7	6	5	4	3	2	1	0
20	0	2	0	0	2	6	4	2	0	9	8	7	6	5	4	3	2	1
21	1	0	1	1	3	0	5	3	1	10	9	8	7	6	5	4	3	2
22	0	1	2	2	4	1	6	4	2	0	10	9	8	7	6	5	4	3
23	1	2	3	3	5	2	7	5	3	1	11	10	9	8	7	6	5	4
24	0	0	0	4	0	3	0	6	4	2	0	11	10	9	8	7	6	5
25	1	1	1	0	1	4	1	7	5	3	1	12	11	10	9	8	7	6
26	0	2	2	1	2	5	2	8	6	4	2	0	12	11	10	9	8	7
27	1	0	3	2	3	6	3	0	7	5	3	1	13	12	11	10	9	8
28	0	1	0	3	4	0	4	1	8	6	4	2	0	13	12	11	10	9
29	1	2	1	4	5	1	5	2	9	7	5	3	1	14	13	12	11	10
30	0	0	2	0	0	2	6	3	0	8	6	4	2	0	14	13	12	11
31	1	1	3	1	1	3	7	4	1	9	7	5	3	1	15	14	13	12
32	0	2	0	2	2	4	0	5	2	10	8	6	4	2	0	15	14	13

Mapas de compresión

- Problemas:
  - Se producen muchas colisiones debido a la reducción del resultado de la función hash con el “mod”.

- Fijar el tamaño de la tabla hash como el doble del necesario para almacenar todos los elementos.
  - Si queremos ampliar el tamaño de la tabla, hay que recalcular las posiciones de todos los elementos e insertarlos en estas nuevas posiciones.
  - NO se puede copiar directamente el contenido de una tabla a otra.

- El tamaño de la tabla debe ser un número primo para evitar bucles infinitos, por la función MOD.

Ejercicio

- Diseñe un procedimiento para calcular el menor valor primo mayor que un valor N.

- Use el postulado de Bertandd's que indica que para un entero n > 3, siempre se cumple, que en el intervalo de [n a 2n – 2], existe al menos un número primo.

```
boolean esPrimo(int n) {
    for(int i=2;i<n;i++) {
        if(n%i==0)
            return false;
    }
    return true;
}
```

Tarea

- Investigar el método de Horner, para calcular el resto de una operación de cálculo de polinomios, sin necesidad de calcular la operación.

- Presentar la corrida de al menor 3 ejemplos .

- Presentar un algoritmo para calcular el mod.

Método de la multiplicación

- Elegir una constante A entre 0 y 1.
- Multiplicar la clase (o el número hash) *k* por A.
  - Se aconseja que A sea (sqrt(5) -1)/2 = 0.6180..
- Extraer la parte fraccionaria de *k**A.
- Multiplicar la parte fraccionaria por *m* (el m no es un valor crítico y se puede calcular como una potencia de 2).
- Tomar el redondeo hacia abajo como resultado.
- **Desventaja:** Es más lento que el método de división.

Método de la multiplicación

A						
0.3629	k*A	frac	m		frac * m	índice
2	0.7259	0.7259	1	2	1.4517	1
3	1.0888	0.0888	2	4	0.3552	0
4	1.4517	0.4517	2	4	1.8069	1
5	1.8147	0.8147	3	8	6.5174	6
6	2.1776	0.1776	3	8	1.4208	1
7	2.5405	0.5405	3	8	4.3243	4
8	2.9035	0.9035	3	8	7.2278	7
9	3.2664	0.2664	4	16	4.2625	4
10	3.6293	0.6293	4	16	10.0695	10
11	3.9923	0.9923	4	16	15.8764	15
12	4.3552	0.3552	4	16	5.6834	5
13	4.7181	0.7181	4	16	11.4903	11
14	5.0811	0.0811	4	16	1.2973	1
15	5.4440	0.4440	4	16	7.1042	7
16	5.8069	0.8069	4	16	12.9112	12
17	6.1699	0.1699	5	32	5.4362	5
18	6.5328	0.5328	5	32	17.0501	17
19	6.8958	0.8958	5	32	28.6640	28
20	7.2587	0.2587	5	32	8.2779	8

Ejercicio

- Elabore el Método de la Multiplicación en una hoja de cálculo.

TABLA DE HASH

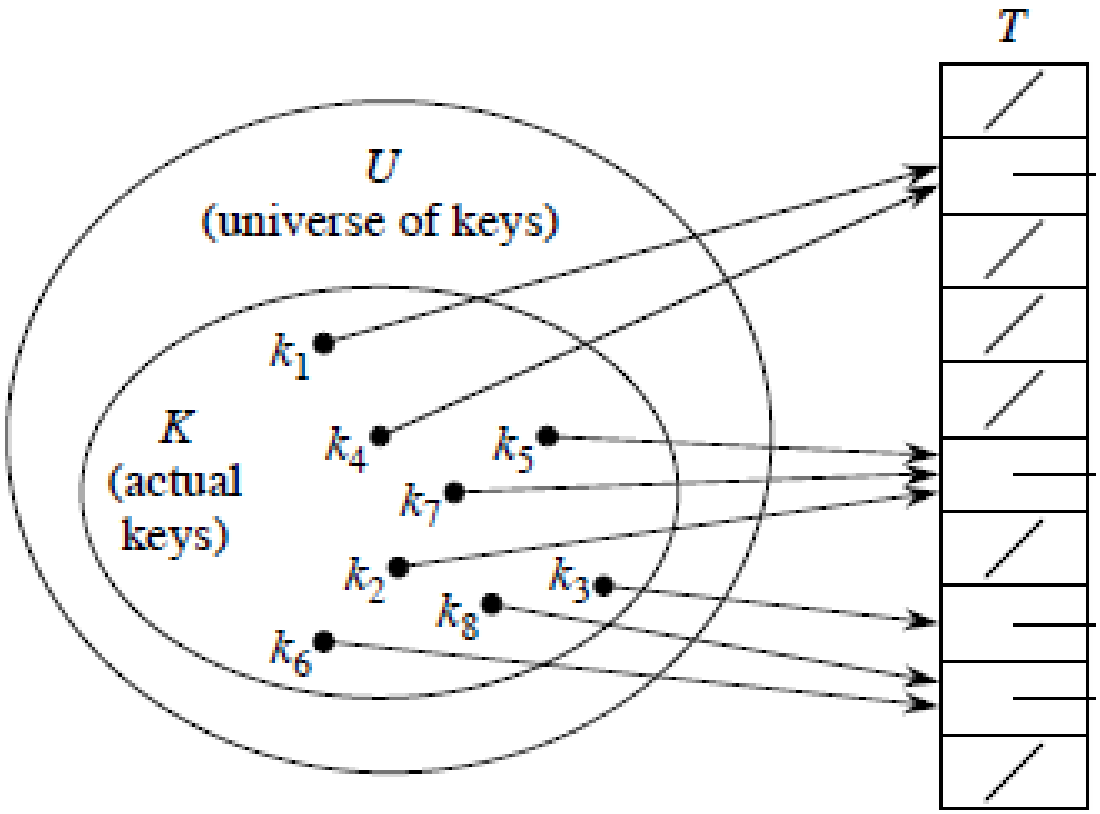
Funciones de dispersión

Mapas de compresión

Gestión de las colisiones

Colisiones

- Una colisión se genera cuando más de una llave apunta a un mismo dato después de aplicarle la función hash.



Gestión de las colisiones

- Es posible que para dos claves la función de *hash* produzca el mismo resultado **(colisión)**.

$$\textit{Colisión:} \quad k_1 \neq k_2 \ ; \quad h(k_1) = h(k_2)$$

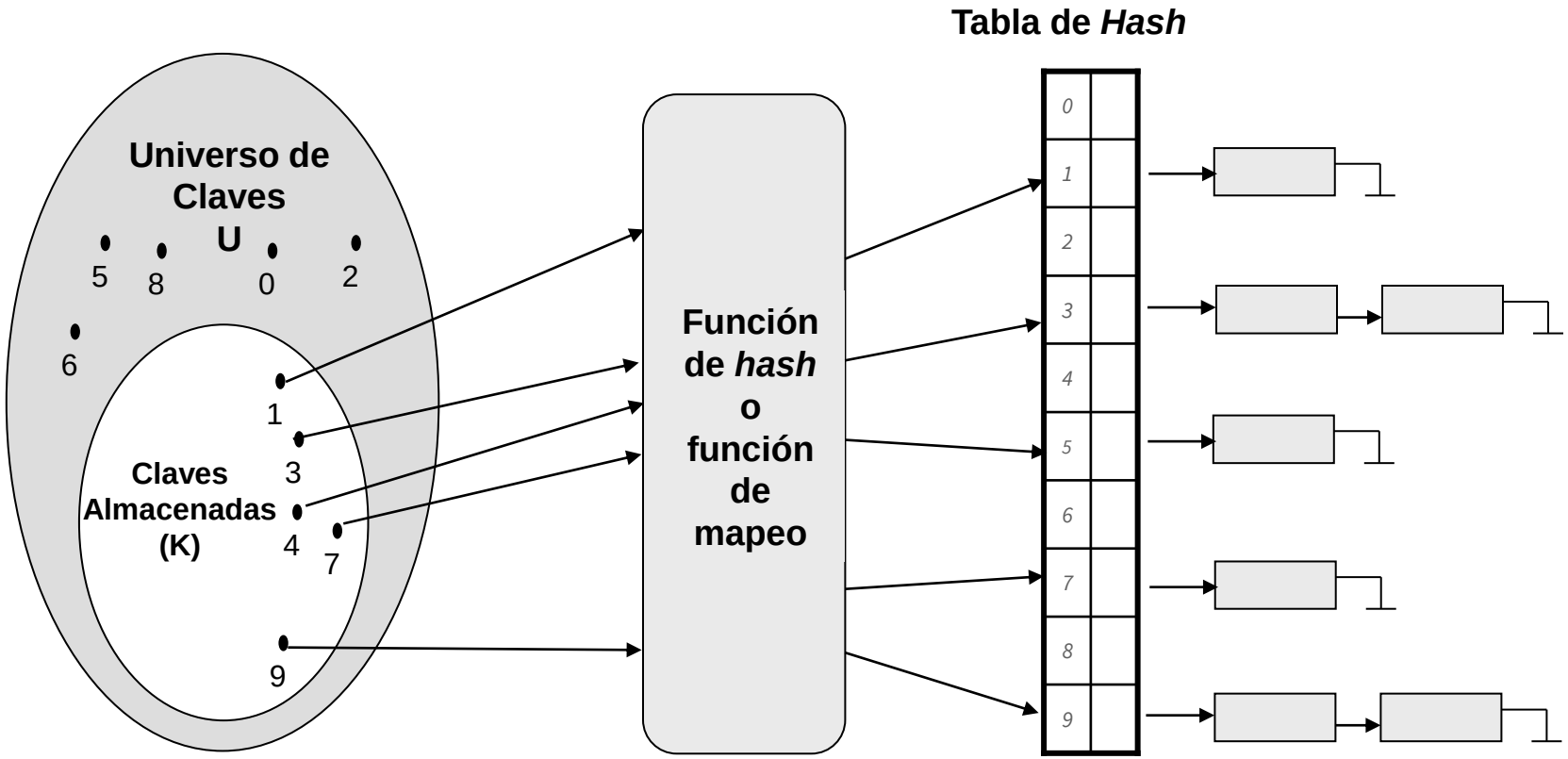
- **Existen dos estrategias:**
- ***Hashing* Abierto:** Cada entrada de la tabla contiene una lista enlazada en la cual se almacenan todos elementos que, de acuerdo a la función de *hash*, correspondan a dicha posición arreglo.

- ***Hashing* Cerrado:** Cada entrada de la tabla contiene un solo elemento. Cuando ocurre una colisión, ésta se soluciona buscando celdas alternativas hasta encontrar una vacía (dentro de la misma tabla)

Hashing abierto

- También llamada ***encadenamiento separado***, consiste en tener en cada posición de la tabla, una lista de los elementos que de acuerdo a la función de *hash*, correspondan a dicha posición.
- El peor caso de hashing abierto nos conduce a una lista con todas las claves en una única lista. El peor caso para búsqueda es así O(n).

Hashing abierto



Hashing cerrado

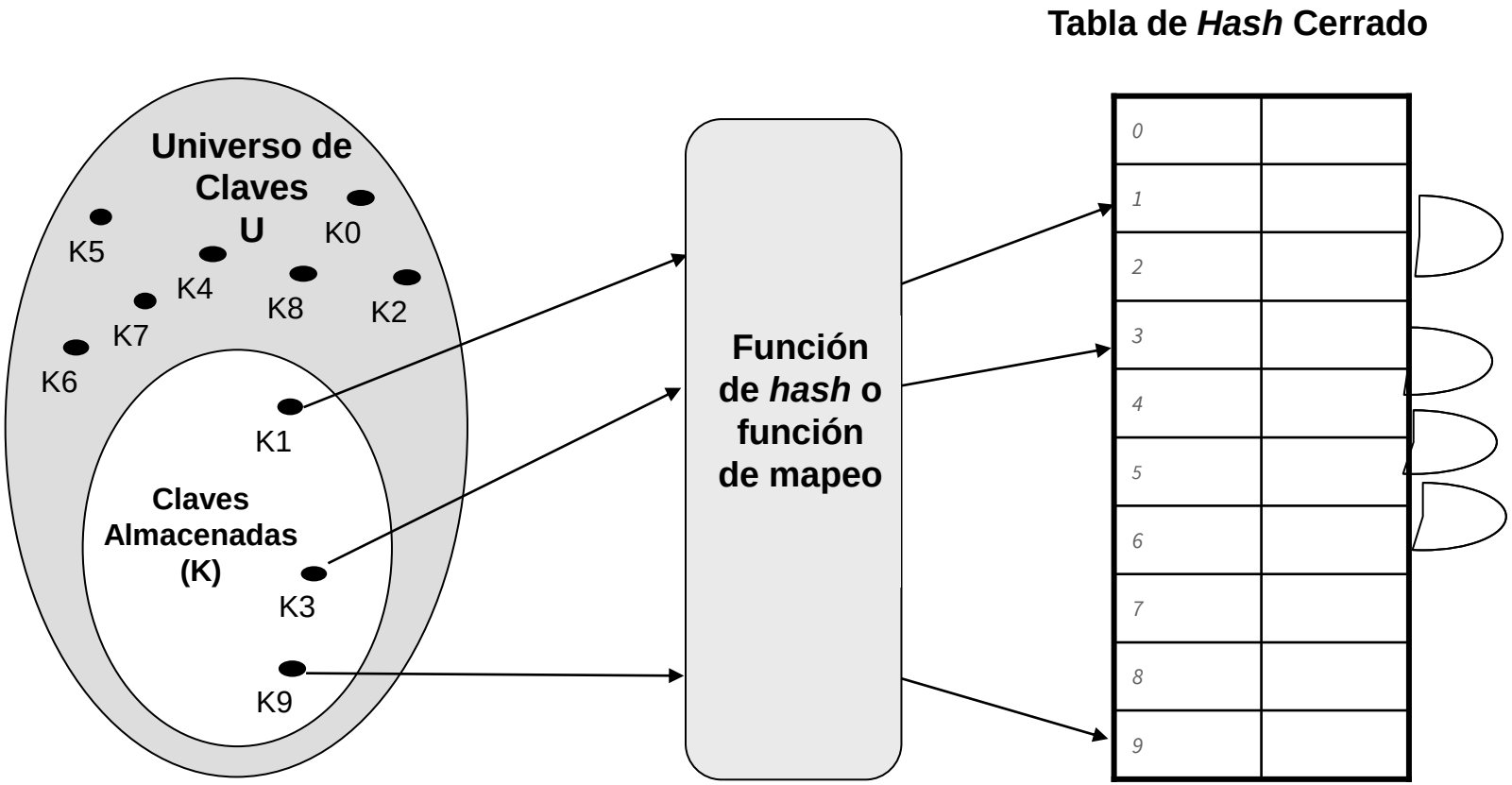
- También llamada ***direccionamiento***, soluciona las colisiones buscando celdas alternativas hasta encontrar una vacía (dentro de la misma tabla).

- Se va buscando en las celdas: $d_0(k)$, $d_1(k)$, $d_2(k)$, **...**, donde
  - $d_i(k) = (h(k) + f(i)) \bmod MAX\_TABLA$
  - $h(k, i) = (h'(k) + f(i)) \bmod MAX\_TABLA$

- La inserción se efectúa probando la tabla hasta encontrar un espacio vacío.

- Para buscar una clave, se examinan varias celdas de la tabla hasta que se encuentra la clave buscada, o es claro que está no está almacenada.

Hashing cerrado



Hashing cerrado

Ventajas

- Elimina totalmente los apuntadores. Se libera así espacio de memoria, el que puede ser usado en más entradas de la tabla.

Hashing cerrado

Desventajas

- Si la aplicación realiza eliminaciones frecuentes, puede degradarse el rendimiento.
- Para garantizar el funcionamiento correcto, se requiere que la tabla de *hash* tenga, por lo menos, el **50% del espacio disponible**.

Tipos de direccionamiento

- Mejorar el direccionamiento a través de:
  - Linear probing (Prueba Lineal)
    - Intervalo de crecimiento constante.
    - $h(k,i) = (h'(k) + i) \bmod m$
  - Quadratic probing (Prueba Cuadrática)
    - Índices descritos por una función cuadrática.
    - $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - Double hashing (Doble hashing)
    - Intervalo calculado por otro hash
    - $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

Prueba lineal

- La función es:
  $h(k,i) = (h'(k) + i) \bmod m$

- Una desventaja de este método es la tendencia a crear largas secuencias de entradas ocupadas, incrementando el tiempo de inserción y búsqueda.

Ejercicio

- Usando la siguiente función de hash cerrada:
- h’(k) = k mod 19

- h(k,i) = (h’(k) +i) mod 29

- Genere los índices para los siguientes números:

- 6, 34, 67, 92, 96, 8, 5, 3, 2

Prueba cuadrática

- Usa la siguiente función.
- Índices descritos por una función cuadrática.
- $h(k,i) = (h'(k) + c1i + c2i^2 \ ) \bmod m$

Ejercicio

- Usando la función cerrada:
- h’(k) = k mod 19

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

- Para c1 = 0, c2 = 1

- m = 31

- Indexe los siguientes números:

- 6, 27, 9, 65, 31, 35, 12, 21, 3, 2

Doble hashing

- La función es:
  $h(k,i) = (h_1(k) + i*h_2(k)) \bmod m$

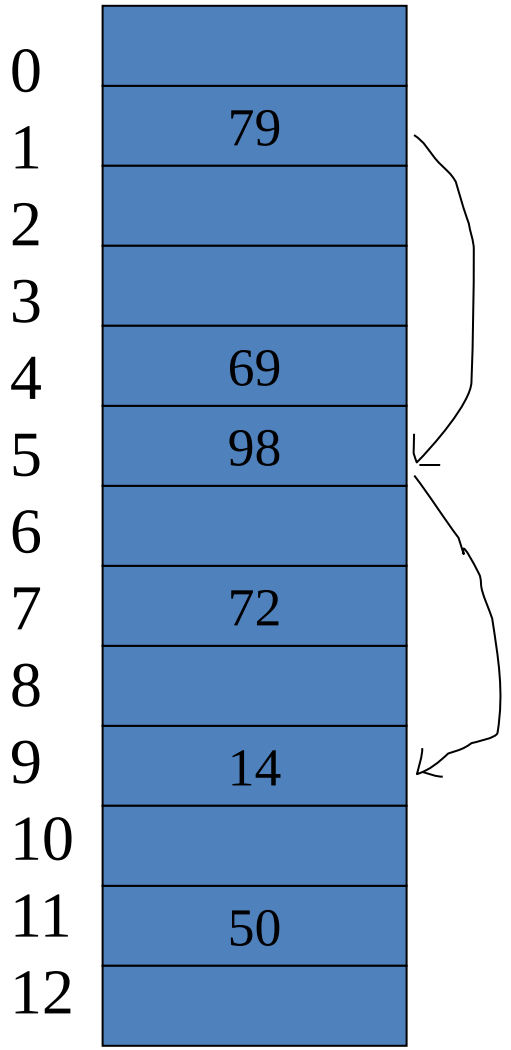
- Por ejemplo:
  $h_1 = k \bmod m$
  $h_2 = 1 + (k \bmod (m-1))$

Ejercicio

- Inserte los siguientes número en la tabla de hash, usando las siguientes funciones de hash (hash cerrado)

- k = 79, 72, 98, 14

- Sea $h(k,i) = (h_1(k) + i*h_2(k)) \bmod 13$; con
  $h_1 = k \bmod 13$
  $h_2 = 1 + (k \bmod 11)$



Ejercicio

- h(79,0) = 1
  h(72,0) = 7
  h(98,0) = 7
  h(98,1) = (7+11) mod 13 =5
  h(14,0) = 1
  h(14,1) = (1+4) mod 13 = 5
  h(14,2) = (1+2*4) mod 13 = 9

Ejercicio

- h1(k) =k mod 13
- h2(k) = 1 + k mod 17
- h(k, i) = [h1(k) + 2 i + 3 i^2 + i h2(k)] mod 19

- 23
- 10
- 31
- 45
- 22
- 23
- 9
- 11

Ejercicio

- Supongamos que queremos guardar los datos de una clase de estudiantes ingresantes en una tabla de dispersión. Implemente una función de hasing para ello. Asuma los datos básicos de cada estudiante.

Investigar

- Fraccionamiento de la tabla hash.
- Es posible que la tabla hash se fracciones cuando constantemente se le inserta y elimina elementos.
- ¿Indicar cuál es el límite?

PREGUNTAS