Estructuras de Datos

Dr. Martin Gonzalez-Rodriguez

ISBN 978-1-365-00700-2

Tablash Hash

Dr. Martin Gonzalez-Rodriguez

Estructuras de Datos Diccionario

Objetivo

- Almacenar objetos sin relaciones entre sí para su recuperación de la manera más rápida posible.
  - Máxima velocidad de acceso.
  - Usan grandes cantidades de memoria.
  - Ampliamente utilizadas en **sistemas de caché en la web** y acceso a **bases de datos**.

Estructuras de Datos Diccionario

Objetivo

- Eficiencia temporal O(1) en operaciones de acceso
  - Se resiente la eficiencia en el resto de las operaciones.

Método	Complejidad
Insertar	O(1)
Buscar	O(1)
Borrar	O(1)
print	O(n)
Obtener Máximo	O(n)
Obtener Mínimo	O(n)

Tablas Hash

Componentes Básicos

HashTable class

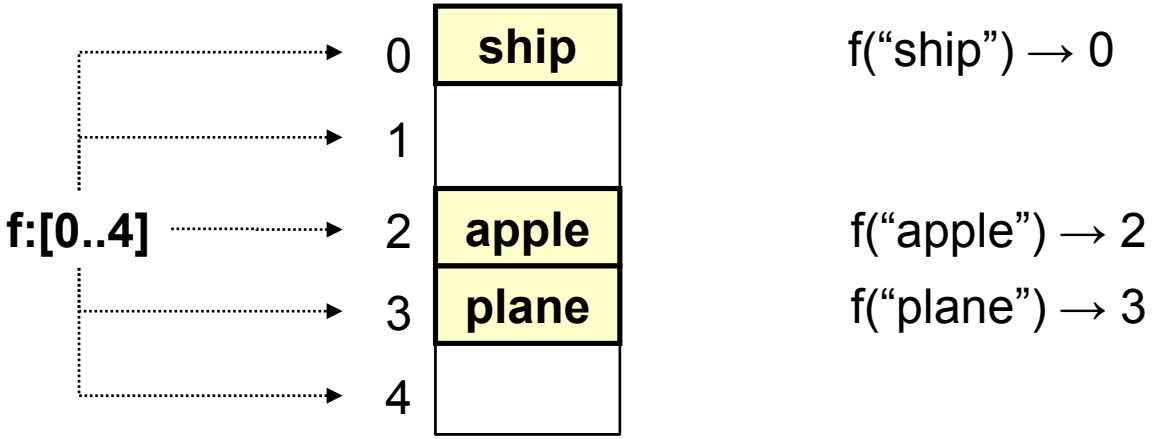
```
class HashTable<T> {
  private final static int B = 5; //size
  private ArrayList<HashNode<T>> associativeArray;

  public HashTable() {
        associativeArray = new ArrayList<HashNode<T>>(B);
  }

  int private f (T element){
        return (…);     // converts T to an int value in the range
                        // [0, B-1].
  }
}
```

Función Hash

Convierte claves en índices

- Recibe la clave de un objeto en el dominio del problema.
  - Usualmente *String* o *int*.

- Devuelve la posición en la que debería alojarse el elemento en el *associativeArray*.
  - Rango de f: [0, B-1].

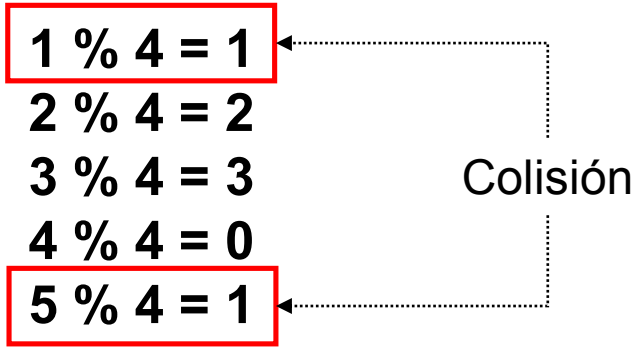


Función Hash

Función f para claves enteras

```
private int f (T element)
{
  return (element.hashCode() % B);
}
```

- Es una operación fácil y rápida de ejecutar.
  - Si las claves son aleatorias, distribuye los elementos uniformemente.



Función Hash

Colisiones

- Dos elementos x e y son **sinónimos** si…
  - f(x) == f (y)
  - Los elementos **sinónimos** producen **colisiones** sobre la misma posición del vector.

- Tratamiento de las colisiones:
  - **Protección Activa**
    - Evitar la colisión (diseño de la función hash perfecta).
  - **Protección Pasiva**
    - Dos o más elementos comparten la misma posición del vector.
  - **Redispersión**
    - Aumentar o disminuir el tamaño del vector (B) de forma dinámica en base al número de elementos que contiene.

Función Hash

Función f perfecta
$P(f(X_1)=0) == P(f(X_2)=1) = \ldots == P(f(X_m)=B-1) == 1/B$

- Garantiza el menor número de colisiones posibles.
  - Por **cada n elementos** a insertar, tan **sólo** se producirían **n/B** colisiones.

10 % 10 = 0
20 % 10 = 0
30 % 10 = 0
40 % 10 = 0
50 % 10 = 0

10 % 7 = 3
20 % 7 = 6
30 % 7 = 2
40 % 7 = 5
50 % 7 = 1

- ¡B **debería ser** un número primo!
  - Ayuda a reducir colisiones cuando las claves **no son aleatorias**.

Función Hash

HashCode para claves String (Versión 1)

```
public int convert (String t){ // <-> t.hashCode()
 int result = 0;

 for (int i=0; i<t.length(); i++)
  result += (int) t.charAt(i);

 return (result);
}

private int f (String element)
{
  return (convert(element) % B);
}
```

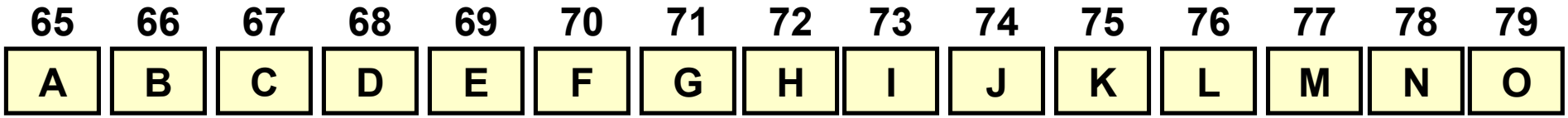
- Convierte la cadena a un entero para luego aplicar la función de dispersión.
  - La función *convert* suma los códigos de representación de cada letra de la cadena.
    - (Códigos ASCII, EBDIC, etc).

Función Hash

Ejercicio

- Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código
P	80
L	76
A	65
N	78
E	69
Total	**368**



Función Hash

Ejercicio

- Calcular el rango de f suponiendo…
  - Cadenas de una longitud máxima igual a 8 caracteres.
  - Rango de códigos [0, 127].
  - B igual a 10.007 posiciones.

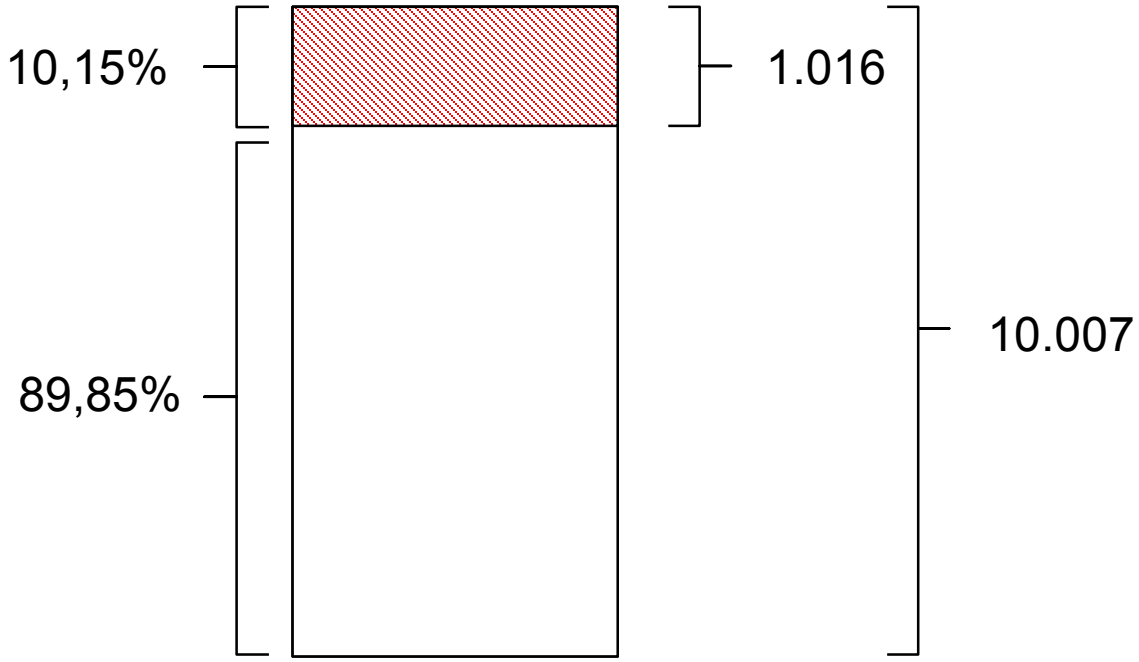
Rango *convert (String t)*
`[8*0, 8*127] = [0, 1.016]`

Rango *f (String t)*
`[0, 1.016] % 10.007 = [0, 1.016]`

Función Hash

Desventajas

- Si **B es grande** y la **longitud de la clave es pequeña**, la dispersión se concentrará en la zona superior del vector.
  - Si la longitud es pequeña, la suma de los códigos también lo será.
  - Al aplicar el operador módulo (%) entre la pequeña suma y un valor de B grande, el resultado obtenido será muy pequeño.



Función Hash

HashCode para claves String (Versión 2)

```
public int convert (String t){// <-> t.hashCode()
 int result = 0;
 int k =(t.length()>3)?3:t.length();

 for (int i=0; i<k; i++)
  result += (int) Math.pow(27, 2-i) * (int) t.charAt(i);

 return (result);
}
```

- Asigna un peso a cada carácter en función de su posición.
  - El valor del peso (27) se corresponde con la longitud del alfabeto.
    - La ponderación es 27^{2-i} siendo i la posición del carácter en la cadena.
  - Se puede restringir el número de caracteres analizados a un límite máximo k por razones de eficiencia.
    - En el ejemplo, k <= 3.
    - La operación de multiplicación consume mucho tiempo de CPU.

$$\text{Convert (“PLANE”)} = P * 27^2 + L * 27^1 + A * 27^0$$

Función Hash

Ejemplo

- Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código Ponderado	Total
P	$80*27^2$	58.320
L	$76*27^1$	2.052
A	$65*27^0$	65
N	-	-
E	-	-
Total		**60.437**

60.437 % 10.007 = 395

La versión 1 de *Convert(“PLANE”)* obtenía 358 (358 % 10.007) = 358

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Función Hash

Desventajas

- Palabras que empiezan con la misma combinación de letras conducen a colisiones.
  - "**PLA**NE", "**PLA**NING", "**PLA**STIC", etc.

- Suponiendo un vector de tamaño B = 10.007…
  - En **Teoría**…
    - Para k=3 existen 27*26*25 (17.550) combinaciones distintas de inicio de palabra en el dominio de la función *convert*.
    - Dado que 17.550 > 10.007, los elementos se distribuyen por todo el vector.
  - En la **Práctica**…
    - De las 17.550 combinaciones posibles solo tienen sentido 2.851 en lengua castellana.
      - » Por ejemplo, no existen palabras que empiecen por ZYV, ZVW, XYV, etc.
    - Con 2.851 palabras válidas **tan solo se emplea un 28,4%** de las 10.007 posiciones disponibles en el vector.

Se hace necesario explorar todos los caracteres de la cadena

Función Hash

HashCode para claves String (Versión 3)

```
public long convert (String t){ // <-> t.hashCode()
 long result = 0;

 for (int i=0; i<t.length(); i++)
  result += (int) Math.pow(32, t.length()-i-1) * (int) t.charAt(i);

 return (result);
}
```

- ¿Cómo optimizar la función para analizar toda la cadena?
  - Se **utiliza 32 como peso**, en lugar de 27.
    - A nivel binario, multiplicar por 32 equivale a un desplazamiento de 5 bits (operación mucho más rápida que una multiplicación).
      - $32 = 2^5$.

$$\text{Convert}(\text{“PLANE”}) = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

DEBATE: ¿Cómo reducir el número de llamadas a la función potencia?

Función Hash

HashCode para claves String (Versión 4)

```
public long convert (String t){// <-> t.hashCode()
 long result = (int) t.charAt(0);

 for (int i=1; i<t.length(); i++)
  result = (32 * result) + (int) t.charAt(i);

 return (result);
}
```

- Utilización de la **Regla de Horner**
  - Minimiza el uso de las multiplicaciones utilizando una representación alternativa del polinomio.

$$\text{Convert (“PLANE”)} = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

$$\text{Convert}_{\text{Horner}} \text{ (“PLANE”)} = ((((P * 32) + L) * 32) + A) * 32) + N) * 32 + E$$

Función Hash

HashCode para claves String (Versión 5)

```
public long convert (String t) {// <-> t.hashCode()
 long result = (int) t.charAt(0);

 for (int i=0; i<t.length(); i++)
  result = ((32 * result) + (int) t.charAt(i)) % B;

 return (result);
}
```

- Eliminación del *Overflow*
  - Durante el cálculo se generan **cifras tan grandes que no puedan ser almacenadas**.
  - Se debe aplicar el operador **resto** (**%)** en cada iteración para reducir el tamaño de las cifras parciales
    - Se elimina el *overflow* a costa de una penalización temporal.

f(“PLANE”) =

(((((P * 32) + L) % B * 32) + A) % B * 32) + N) % B * 32 + E) % B

Función Hash

Ejemplo

- Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código Ponderado	Total
P	$80*32^4$	83.886.080
L	$76*32^3$	2.490.368
A	$65*32^2$	66.560
N	$78*32^1$	2.496
E	$69*32^0$	69
Total		**86.445.573**

86.445.573 % 10.007 = 5.107

La versión 2 de *Convert(“PLANE”)* obtenía 60.437 % 10.007 = 395

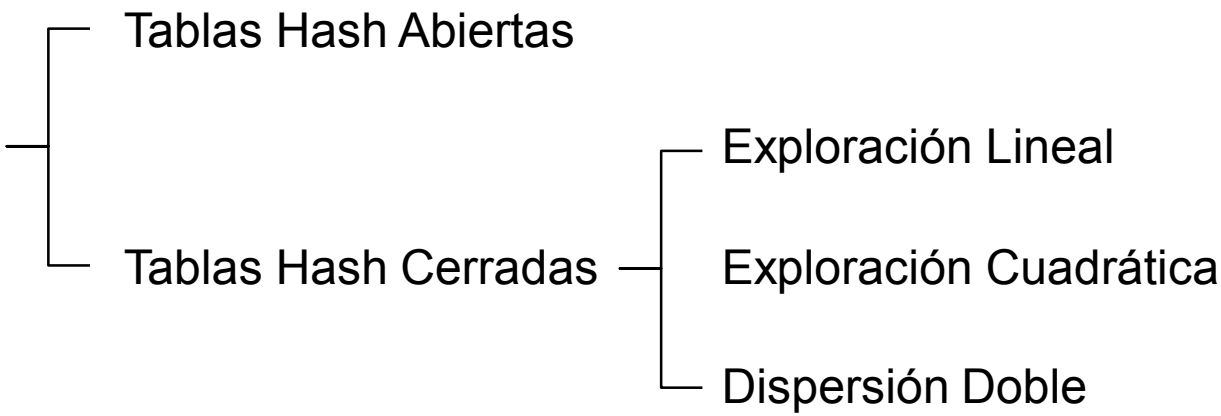
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Protección Pasiva

Las Colisiones son inevitables a largo plazo…

- Cuanto **menor** sea B **mayor** será la probabilidad de colisión.
  - La certeza se alcanza con…
    - B = 1.
    - Dominios de problema en los que existen elementos de clave repetida.

- Dos o más elementos comparten la misma posición del vector.
  - Existen varias formas de gestionar elementos en colisión.



Tablas Hash Abiertas

Tablas Hash Abiertas

- Cada celda contiene una estructura de datos dinámica encargada de almacenar los sinónimos.
  - LinkedList.
  - AVLTree.

HashTable class **O(B) = O(1)**

```
public class HashTable<T>
{
 private final static int B = 10007;
 private AVL<T> associativeArray[];

 public HashTable(int B) {
  this.B = B;
  associativeArray = new AVL<T>[B];

  for (int i=0; i<associativeArray.length; i++)
   associativeArray[i] = new AVL<T>();
 }
}
```

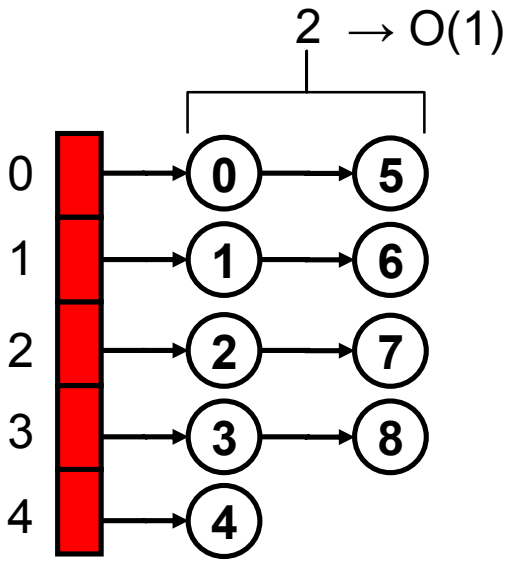
DEBATE: ¿Cuál es la complejidad Temporal del constructor?

Tablas Hash Abiertas

add	O(n/B) → O(1)

```
public void add (T a){
 if (!find(a))
  associativeArray[f(a.hashCode())].add(a);
}
```

find() y remove() son análogas

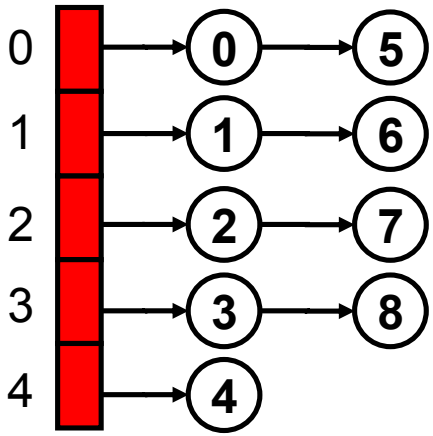



```
for (int i=0; i< 9; i++)
 table.add(new Integer(i), i);
```

Tablas Hash Abiertas

Factor de Carga (load factor)

- Número de elementos de la tabla dividido entre el tamaño de la tabla.
  - LF = n/B.
  - Coincide con la longitud media de cada lista.



LF = 9/5 = 1,8

Tablas Hash Abiertas

LF Eficiente

Búsqueda	Promedio de enlaces visitados
Infructuosa	LF
Exitosa	1 + LF/2

- Para garantizar un alto rendimiento en tablas hash abiertas, el LF **ha de ser menor o igual que uno (LF <= 1)**
  - B = n (aproximadamente).
  - Longitud media de las listas = 1.

Tablas Hash Cerradas

Tablas Hash Cerradas

- Cada celda tiene capacidad para un único objeto.
  - Cuando se detecta una colisión (celda previamente ocupada), se busca el elemento en las celdas próximas.
  - Existen diversos enfoques para realizar la exploración:
    - Exploración Lineal.
    - Exploración Cuadrática.
    - Dispersión Doble.

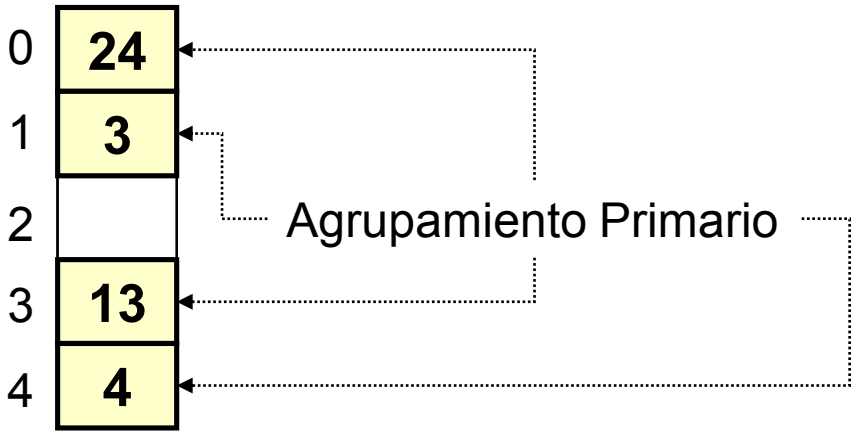
HashTable class

```
public class HashTable<T>
{
  private final static int B = 10007;
  private ArrayList<HashNode<T>> associativeArray;
}
```

Tablas Hash Cerradas

Exploración Lineal

- Búsqueda consecutiva en celdas próximas modificando la función f.
  - f(x) = [x + i] % B.
    - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3…



add(4) → [4 + 0] % 5 = 4

add(13) → [13 + 0] % 5 = 3

add(24) → [24 + 0] % 5 = 4

add(24) → [24 + 1] % 5 = 0

add(3) → [3 + 0] % 5 = 3

add(3) → [3 + 1] % 5 = 4

add(3) → [3 + 2] % 5 = 0

add(3) → [3 + 3] % 5 = 1

Tablas Hash Cerradas

Agrupamientos

- Bloques de celdas ocupadas interrelacionadas.
  - Incluso en tablas relativamente vacías se pueden presentar agrupamientos.
  - Cualquier clave que se disperse sobre un agrupamiento **requerirá varios intentos para su encontrar su ubicación**.
    - Y lo que es peor… si se inserta **se unirá al agrupamiento**.

- Si la tabla es suficientemente grande, se podrá encontrar una ubicación para el elemento…
  - …Pero la búsqueda puede llevar mucho tiempo.

Búsqueda	**Número aproximado de intentos**
Infructuosa	$(1 + 1/(1 – LF)^2)/2$
Exitosa	$(1 + 1/(1 – LF))/2$

Tablas Hash Cerradas

Estudios teóricos de velocidad de acceso

LF	Intentos por Inserción (promedio)
0,90	50
0,75	8,5
0,50	2,5

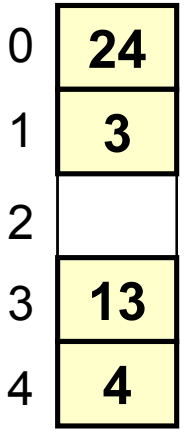
- Se recomienda usar LF <= 0,5
  - B debería ser al menos el doble de n.
  - El incremento en el consumo de memoria es notable.

En las tablas hash abiertas la recomendación es LF <= 1

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

- La existencia de agrupamientos impide el borrado directo de un elemento.
  - El elemento se **marca para borrar** pero no se elimina definitivamente **hasta que su espacio no sea requerido** por una operación de inserción.
  - Los elementos marcados se consideran vacíos durante las inserciones y ocupados durante las búsquedas.



delete(24) → [24 + 0] % 5 = 4

delete(24) → [24 + 1] % 5 = 0

find(3) → [3 + 0] % 5 = 3

find(3) → [3 + 1] % 5 = 4

find(3) → [3 + 2] % 5 = 0

El acceso a la clave 3 se ha perdido

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

HashTable class

```
public class HashNode <T>
{
    public final static byte EMPTY   = 0;
    public final static byte VALID   = 1;
    public final static byte DELETED = 2;

    private T element;
    private byte status = EMPTY;
}
```

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

HashTable class

```
public class HashTable<T>
{
  private final static int B = 10007;
  private ArrayList<HashNode<T>> associativeArray;
}
```

Antes

0	24	DELETED
1	3	VALID
2		EMPTY
3	13	VALID
4	4	VALID

delete(24) → [24 + 0] % 5 = 4

delete(24) → [24 + 1] % 5 = 0

find(3) → [3 + 0] % 5 = 3

find(3) → [3 + 1] % 5 = 4

find(3) → [3 + 2] % 5 = 0

find(3) → [3 + 3] % 5 = 1

add(15) → [15 + 0] % 5 = 0

Después

0	15	VALID
1	3	VALID
2		EMPTY
3	13	VALID
4	4	VALID

Tablas Hash Cerradas

Exploración Cuadrática

- Si se produce una colisión se exploran las celdas a una distancia cuadrática de la anteriormente consultada.
  - $f(x) = [x + i^2] \% B$.
    - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3…

0	**24**
1	
2	**3**
3	**13**
4	**4**

$add(4) \rightarrow [4 + 0^2] \% 5 = 4$

$add(13) \rightarrow [13 + 0^2] \% 5 = 3$

$add(24) \rightarrow [24 + 0^2] \% 5 = 4$

$add(24) \rightarrow [24 + 1^2] \% 5 = 0$

$add(3) \rightarrow [3 + 0^2] \% 5 = 3$

$add(3) \rightarrow [3 + 1^2] \% 5 = 4$

$add(3) \rightarrow [3 + 2^2] \% 5 = 2$

Tablas Hash Cerradas

Exploración Cuadrática

- Puesto que la longitud de los saltos es mayor (longitud cuadrática) es posible no encontrar una posición libre.
  - ¡Aún cuando puedan existir posiciones libres, la exploración cuadrática puede saltar por encima de ellas… ignorándolas!

Teorema de la Exploración Cuadrática

Si utilizando exploración cuadrática **se cumple que** B es primo y el LF <= 0,5 **siempre es posible** encontrar una posición para insertar un elemento.

- La exploración cuadrática elimina el agrupamiento primario…
  - … aún cuando puede crear agrupamientos secundarios.

- El agrupamiento secundario podría llegar a ser asumible…
  - Estudios de simulación demuestran que ante agrupamientos secundarios **tan solo es necesario un salto** para encontrar posiciones libres.

Tablas Hash Cerradas

Dispersión Doble

- Utilizada una doble función de dispersión.
  - $f(x) = [x + i*H_2(x)] \% B$.
    - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3…
    - Donde H_2 es la función de cálculo de salto. Puede ser cualquiera. Se recomienda:
      - $H_2(x) = R – X \% R$.
        - Donde R es el número primo antecesor de B.

0	
1	**3**
2	**24**
3	**13**
4	**4**

add(4) → [4 + 0*(3 – 4%3)] % 5 = 4

add(13) → [13 + 0*(3 – 13%3)] % 5 = 3

add(24) → [24 + 0*(3 – 24%3)] % 5 = 4

add(24) → [24 + 1*(3 – 24%3)] % 5 = 2

add(3) → [3 + 0*(3 – 3%3)] % 5 = 3

add(3) → [3 + 1*(3 – 3%3)] % 5 = 1

Solución: posiciones 4, 1, 3 y 0

Tablas Hash Cerradas

Evaluación de la Dispersión Doble

- Ventajas
  - Elimina el agrupamiento.
  - El número esperado de intentos es bajo.

- Desventajas
  - El uso de una segunda función aumenta el cálculo del coste de ejecución.

Redispersión

Duplicar el tamaño de la tabla dinámicamente

- Si el LF aumenta demasiado…
  - El rendimiento de la tabla decrece considerablemente.
    - LF > 1 en tablas hash abiertas.
  - Se paraliza el funcionamiento de tablas cerradas al no encontrar posiciones libres.
    - LF > 0,5 en tablas hash cerradas.

- La redispersión recupera un LF aceptable **trasladando** los elementos a una tabla de mayor tamaño.
  - Se establece un número B para la nueva tabla buscando el **número primo inmediatamente superior al doble** del original.
  - Recorre secuencialmente los elementos de la tabla original añadiéndolos a la nueva tabla.

Redispersión

Ejercicio

- Redispersar utilizando Exploración Cuadrática

El número primo inmediatamente superior al doble de 5 es el 11

0	24
1	
2	3
3	13
4	4

add(24) → [24 + 0^2] % 11 = 2

add(3) → [3 + 0^2] % 11 = 3

add(13) → [13 + 0^2] % 11 = 2

add(13) → [13 + 1^2] % 11 = 3

add(13) → [13 + 2^2] % 11 = 6

add(4) → [4 + 0^2] % 11 = 4

O(n)

0	
1	
2	24
3	3
4	4
5	
6	13
7	
8	
9	
10	

Redispersión

Activación de la Redispersión

- La redispersión se puede lanzar automáticamente cuando…
  a) Se alcance un LF > 0,5.
  b) Falle una inserción (no hay posiciones libres).
  c) Cuando se supere un cierto umbral de LF definido en el constructor de la tabla hash.

- Redispersión Inversa
  - Reduce el tamaño de la tabla para ahorrar memoria cuando se han realizado muchas operaciones de borrado.

Tipo de tabla	Umbral de LF para redispersión inversa
Abierta	0,33
Cerrada	0,16

HOMEWORK

PLAYGROUND

- Consulte la entrada para la **Tabla Hash** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Ponga especial atención a las situaciones en las que resulta interesante utilizar árboles en lugar de listas en las tablas hash abiertas.
    - Analice en detalle las funciones de dispersión *Hash de División* y *Hash de Multiplicación*.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Apéndices

Referencias

Referencias

BRASSARD G.; BRATLEY, P.; (1997) *Fundamentos de Algoritmia*. Prentice Hall. ISBN: 84-89660-00-X. [Cap. 5].

COLLADO M., MORALES R. y MORENO J. (1987) Estructuras de datos. Realización en Pascal. Ed. Díaz de Santos, 1987.

WEISS, Mark Allen; (2000) *Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN84-7829-035-4. [Cap. 19].

WEISS, Mark Allen (1995) *Data Structures and Algorithm Analysis*. Addison-Wesley Iberoamericana. [Cap. 5].