Estructura de Datos y Algoritmos 1

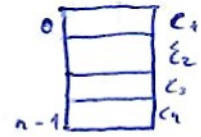
Tema 4 - Tablas Hash

Hash: Dispersión

Historia

LINEAL $O(n)$ — ∞ — ÁRBOLES $O(\log n)$

TABLAS HASH $\sim O(1)$ → TABLA



c_i: Posición dentro de la tabla
Clave numérica: Índice de la tabla

Si la clave es String → hashCode

Tablas de acceso directo → Se utiliza la clave para indicar la tabla.

Función que elijo → Inyectiva

Tablas de acceso directo

Exige clave numérica
Transformación inyectiva

Tablas Hash

n elementos

$H(c_i)$ → pos

H → NO tiene por qué ser inyectiva

$H(c_1) = H(c_2)$

Ocurre una colisión

c_1 y c_2 → Sinónimos

c_1 y c_2 ∈ Clases de equivalencia.

Función hash más sencilla

$C = C_0, C_1, C_2, C_3, C_4$

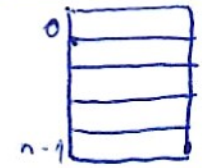
→ $ASCII(C_0) + ASCII(C_1) + \dots + ASCII(C_4)$

eva → 76

ave → 76

vea → 76

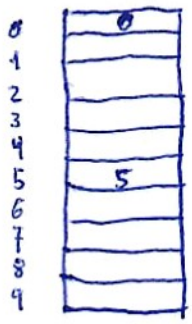
Tamaño N



$H(c) = c \% N$ → Función hash más simple

Ejemplo: $E = \{0, 10, 20, 5\}$

$H(c) = c \% 10$



$H(0) = 0 \% 10 = 0$ ✓

$H(10) = 10 \% 10 = 0$ × Colisión

$H(20) = 20 \% 10 = 0$ × Colisión

$H(5) = 5 \% 10 = 5$ ✓

Si quiero orden lexicográfico, vuelco datos a estructura AVLTree <T>

Cuando el orden NO importa → Tabla Hash.

Ejemplo:

$H(c) = c \% 7$

$E = \{3, 13, 5, 15, 9, 7\}$

Exploración lineal (Gestión de colisiones)

Redimensionamos $n = 11$ (rehashing)

toString() → Ordenado (Lexicográfico

- Exploración lineal:

Función hash (H) → $p(c) = (H + i) \% n$

0	7
1	15
2	9
3	3
4	
5	5
6	13

$P_0(3) = (3 \% 7 + 0) \% 7 = 3 \% 7 = 3$ ✓

$P_0(13) = (13 \% 7 + 0) \% 7 = 13 \% 7 = 6$ ✓

$P_0(5) = (5 \% 7 + 0) \% 7 = 5 \% 7 = 5$ ✓

$P_0(15) = (15 \% 7 + 0) \% 7 = 15 \% 7 = 1$ ✓

$P_0(9) = (9 \% 7 + 0) \% 7 = 9 \% 7 = 2$ ✓

$P_0(7) = (7 \% 7 + 0) \% 7 = 7 \% 7 = 0$ ✓

- Rehashing: $(n = 11)$

0	
1	
2	13
3	3
4	15
5	5
6	
7	7
8	
9	9
10	

$H(c) = c \% 11$

$P_0(7) = 7 \% 11 = 7$

$P_0(15) = 15 \% 11 = 4$

$P_0(9) = 9 \% 11 = 9$

$P_0(3) = 3 \% 11 = 3$

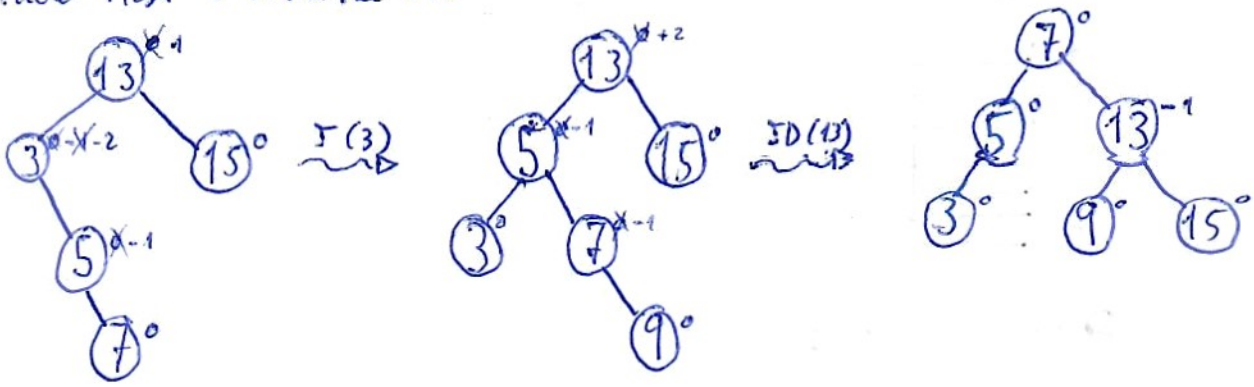
$P_0(5) = 5 \% 11 = 5$

$P_0(13) = 13 \% 11 = 2$

↓ (Orden de la tabla anterior)

- toString Ordenar ()

Table Hash → AVL Tree <T>



Tablas Hash en Java

TreeMap <K, V> ORDEN HashMap <K, V>

TreeSet <T> ORDEN HashSet <T>

T = K + V

↳ Comparable <K>

↳ { int hashCode () {}
boolean equals (Object otro) {}

hashCode Object

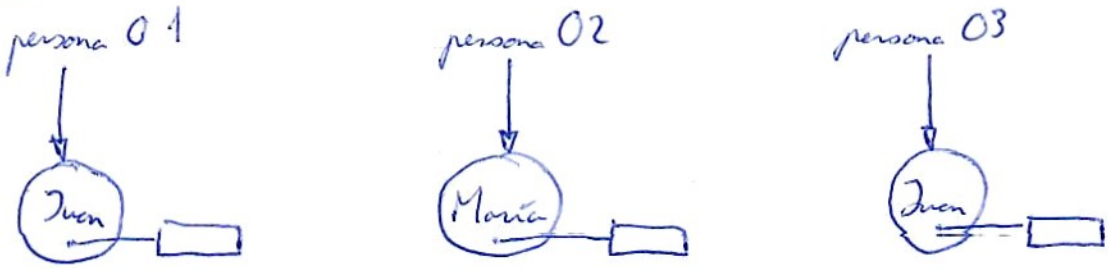
↳ Posición en función de su posición de memoria

HashSet <String> estructura:

↳ HashCode ()

Double / Integer

HashSet <Persona> estructura;



persona01.hashCode() → 37804

persona02.hashCode() → 4896

persona03.hashCode() → ≠ 37804

```
public int hashCode() {
  return atributoClave.hashCode();
}

@Override
public boolean equals (Object otro) {
  // Dos objetos son iguales si sus hashCode lo son.
}

@Override
compareTo()
```

¿Qué ocurre si la clave es compuesta?

clavePrincipal.hashCode()
claveSecundaria.hashCode()
} Objects.hashCode(_, _, _, ..., _)

↳ {hashCode de un conjunto de datos}

¿Por qué se utiliza el 31 en los hash?

1) Número primo → Reduce colisiones

2) Optimización → Puede implementarse como un desplazamiento y una resta.

3) Dispersión de valores → Proporciona buena distribución de valores

Ejemplo 2:

$H_1(c) = c \% 11$. $H_2(c) = 7 - c \% 7$

$P_i(c) = [H_1(c) + i \cdot H_2(c)] \% 11$

$E = \{3, 6, 7, 25, 5, 12, 4, 23, 33, 11\}$

		FLAG
0	23	0
1	12	0
2	33	0
3	3	0
4	4	0
5	5	0
6	6	0
7	7	0
8		L
9	25	0
10	11	0

$P_0(3) = (3 \% 11 + 0 \cdot (7 - 3 \% 7)) \% 11 = 3$ ✓

$P_0(6) = 6 \% 11 = 6$ ✓

$P_0(7) = 7 \% 11 = 7$ ✓

$P_0(25) = 25 \% 11 = 3$ ✗

$H_2(25) = 7 - 25 \% 7 = 3$

$P_1(25) = 3 + 1 \cdot 3 = 6$ ×

$P_2(25) = 3 + 2 \cdot 3 = 9$ ✓

$P_0(5) = 5 \% 11 = 5$ ✓

$P_0(4) = 4 \% 11 = 4$ ✓

$P_0(23) = 23 \% 11 = 1$ ×

$H_2(23) = 7 - 23 \% 7 = 5$

$P_1(23) = 1 + 1 \cdot 5 = 6$ ×

$P_2(23) = 1 + 2 \cdot 5 = 11 = 0$ ✓

$P_0(33) = 33 \% 11 = 0$ ×

$H_2(33) = 7 - 33 \% 7 = 2$

$P_1(33) = 0 + 1 \cdot 2 = 2$ ✓

$P_0(11) = 11 \% 11 = 0$ ×

$H_2(11) = 7 - 11 \% 7 = 3$

$P_1(11) = 0 + 1 \cdot 3 = 3$ ×

$P_2(11) = 0 + 2 \cdot 3 = 6$ ×

$P_3(11) = 0 + 3 \cdot 3 = 9$ ×

$P_4(11) = 0 + 4 \cdot 3 = 12 \% 11 = 1$ ×

$P_5(11) = 0 + 5 \cdot 3 = 15 \% 11 = 4$ ×

$P_6(11) = 0 + 6 \cdot 3 = 18 \% 11 = 7$ ×

$P_7(11) = 0 + 7 \cdot 3 = 21 \% 11 = 10$ ✓

Buscar elemento

¿ Contains (5) ?

$P_0(5) = 5 \% 11 = 5$ Sí está

¿ Contains (33) ?

$P_0(33) = 33 \% 11 = 0$

$H_2(33) = 7 - 33 \% 7 = 2$

$P_1(33) = 0 + 1 \cdot 2 = 2$ Sí está

Eliminamos 23 y buscamos 33

remove (23)

¿ Contains (33) ?

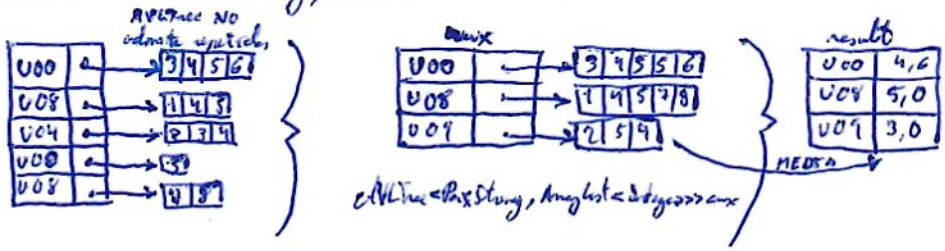
$P_0(33) = 33 \% 11 = 0$ → Como la posición 0 está vacía, concluye que 33 no está

$\lambda = \frac{n}{N} \simeq 0{,}8$ → Rehashing

Cosas de Java

Crear método reduce

```
ArrayList<Par<String, AVLTree<Integer>>> datosOriginal = new ArrayList<>();
AVLTree<Par<String, Double>> datosDestino = new AVLTree();
```




```
public void reduce() {
  AVLTree<Par<String, ArrayList<Integer>>> aux = new AVLTree<>();
  for (Par<String, AVLTree<Integer>> parCurr : datosOriginal()) {
    Par<String, ArrayList<Integer>> parAux = aux.find(new Par<>(parCurr.getKey(), null));
    if (ParAux == null) aux.add(parAux = new Par(parCurr.getKey(), new ArrayList<>()));
    parAux.getValue().addAll(parCurr.getValue());
  }
  for (Par<String, ArrayList<Integer> parAux2 : aux);
    datosDestino.add(new Par<>(parAux2.getKey(), mediaAritmetica(parAux2.getValue())));
  }
}
```

Cosas de Java 2.0

$H(c) = c \% n$

- pos (array tamaño n)
- Gestión de colisiones

$n \rightarrow n_0, n_1, n_2, \ldots, n_{-1}$

Clases de equivalencia (Sinónimos)

DIRECCIONAMIENTO ABIERTO

$P_1(c) = (H(c) + f(i)) \% n$

$$f(i) \begin{cases} f(i) = 0 \\ f(i) = i^2 \\ f(i) = i \cdot H_2(c) \end{cases}$$

ENCADENAMIENTO SEPARADO

ArrayList<AVLTree<T>>

ArrayList<~~PriorityQueue~~<T>>
↳ NO TIENE MÉTODO BÚSQUEDA

TreeSet<~~ArrayList~~<T>>