

ESTRUCTURAS DE DATOS

DICCIONARIOS

# **Tablas *hash* redimensionables**

Manuel Montenegro Montes  
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: factor de carga

- El **factor de carga**  $\alpha$  de una tabla *hash* es el cociente entre el número de entradas en la tabla y el número de cajones.
- Sean:
  - $n$  - número de entradas en la tabla
  - $m$  - número de cajones

$$\alpha = \frac{n}{m}$$

Del vídeo anterior sacamos una conclusión importante: las operaciones en una tabla hash tienen coste constante bajo ciertas suposiciones.

# Tablas redimensionables

- En el caso medio, las operaciones en una tabla hash abierta tienen coste  $O(1 + \alpha)$ .
- Por tanto, conseguimos coste constante en el caso medio si mantenemos el factor de carga acotado.
- Una **tabla hash redimensionable** es una tabla que se amplía cada vez que el factor de carga supera un determinado valor umbral.

208

# Implementación

FACTOR DE CARGA MÁXIMO QUE VAMOS A PERMITIR

```
const int INITIAL_CAPACITY = 31;  
const double MAX_LOAD_FACTOR = 0.8;
```

**Factor de carga máximo permitido**

```
template <typename K, typename V, typename Hash = std::hash<K>>  
class MapHash {  
    ...
```

```
private:  
    using List = std::forward_list<MapEntry>;
```

```
    Hash hash;
```

```
    List *buckets;
```

```
    int num_elems;
```

```
    int capacity;
```

```
    ...  
};
```

**Tamaño del vector**

# Implementación de insert (antes)

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
    ...

    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % capacity;

        auto it = find_in_list(buckets[h], entry.key);

        if (it == buckets[h].end()) {
            buckets[h].push_front(entry);
            num_elems++;
        }
    }

private:
    ...
};
```



# Implementación de insert (después)

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
```

```
...
```

```
void insert(const MapEntry &entry) {
    int h = hash(entry.key) % capacity;
```

```
    auto it = find_in_list(buckets[h], entry.key);
```

```
    if (it == buckets[h].end()) {
```

```
        num_elems++;
```

```
        resize_if_necessary();
```

```
        h = hash(entry.key) % capacity;
```

```
        buckets[h].push_front(entry);
```

```
    }
```

```
}
```

```
private:
```

```
...
```

```
};
```

En el caso de que no se encuentre primero aumentamos el número de elementos

comprueba el factor de carga y si supera la constante anterior, entonces redimensiona la tabla

puede cambiar la capacity

# Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26	14	41		56	18			34		49	37	
-	-	-		-	-			-		-	-	

44  
-

21  
-

5  
-

$$13 * 2 = 26 \quad 24$$

$$n = 11$$

$$m = 13$$

$$\alpha = \frac{1}{13} = 0.0769230769$$

# Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26	14	41		56	18			34		49	37	
-	-	-		-	-			-		-	-	

44  
-

21  
-

5  
-

29

elementos porque 26 NO ES PRIMO

0	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	28



# Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26	14	41		56	18			34		49	37	
-	-	-		-	-			-		-	-	

Volvemos a calcular la función hash para cada uno de ellos.

44  
-

21  
-

5  
-

0	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	28
					5			37				41		14	44			18		49	21					26	56	
					-			-				-		-	-			-		-	-					-	-	

34  
-

# Método auxiliar de redimensionamiento

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
private:
```

```
void resize_if_necessary() {
    double load_factor = ((double)num_elems) / capacity;
    if (load_factor < MAX_LOAD_FACTOR) return;
```

Casteo porque el número de elementos es int.

SI EL FACTOR DE CARGA NO SUPERA EL UMBRAL MÁXIMO

```
    int new_capacity = next_prime_after(2 * capacity);
    List *new_array = new List[new_capacity];
```

Calculamos el siguiente primo más cercano con capacidad mínima 2 por actual capacity

```
    for (int i = 0; i < capacity; i++) {
        for (const MapEntry &entry : buckets[i]) {
            int new_pos = hash(entry.key) % new_capacity;
            new_array[new_pos].push_front(entry);
        }
    }
```

Para cada una de las posiciones del vector

```
    capacity = new_capacity;
    delete[] buckets;
    buckets = new_array;
}
```

Volvemos a calcular el valor hash de esa clave

Inserto al principio en el nuevo array,

```
};
```

# Costes en tiempo

- Suponiendo **dispersión uniforme**  $\alpha < 1$

Operación	Tabla <i>hash</i>
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[]</i>	$O(1) / O(n)$
<i>insert</i>	$O(1) / O(n)$
<i>erase</i>	$O(1)$

ya que en el caso en el que no se encuentre la clave, la inserta

lineal si tenemos que redimensionar

El redimensionado no es frecuente a medida que hacemos esas redimensiones.

$n$  = número de entradas en la tabla

# Costes amortizados en tiempo

- Suponiendo **dispersión uniforme**.

Operación	Tabla <i>hash</i>
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[]</i>	$O(1)$
<i>insert</i>	$O(1)$
<i>erase</i>	$O(1)$

Suponiendo que nuestras funciones hash son buenas.

$n$  = número de entradas en la tabla