

Contenedores Avanzados

-Tablas hash-

T. Hash en STL

unordered_set o unordered_map

unordered_multiset o unordered_multimap (c++11)

- Contenedores asociativos que permiten un acceso rápido a elementos individuales ($O(1)$) (buscar, insertar y borrar).
 - Internamente no existe ningún orden entre los elementos, por lo que al iterar estos se acceden de forma “aleatoria”
 - Son más eficientes que set y map para acceder a los elementos, pero más lentos en la iteración.
 - Solo implementan un iterador de avance.
- unordered_set: almacena elementos

Tablas Hash

- Dada una tabla T y un elemento x , (asumimos x como clave) y conjunto de datos asociados, necesitamos definir las siguientes operaciones:
 - $T.\text{Insertar}(x) \implies O(1)$
 - $T.\text{Borrar}(x) \implies O(1)$
 - $T.\text{Buscar}(x) \implies O(1)$
- **No es importante el orden entre los elementos**
 - Necesitamos que estas operaciones se puedan implementar de forma rápida,
 - Asumimos que todas las claves son numero naturales (posiblemente muy grandes)
 - Que pasa si la clave es alfabetica?

Acceso Directo

- Suponemos:
 - El rango de valores que toma la clave es $0..m-1$
 - No existen claves repetidas (todas distintas)
- Idea:
 - Utilizar un vector $T[0..m-1]$ en el que
 - $T[i] = x$ si $x \in T$ y $\text{clave}[x] = i$
 - $T[i] = \text{NULL}$ en caso contrario
 - Es una *tabla de acceso directo*
 - Las operaciones son $O(1)$, la tabla de tamaño m !!!!
 - Qué pasa si insertamos 10000 DNIs?
 - ◆ Necesitamos una tabla de tamaño 100.000.000!!!

Acceso Directo

- Suponemos:
 - El rango de valores que toma la clave es $0..m-1$
 - No existen claves repetidas (todas distintas)
- Idea:
 - Utilizar un vector $T[0..m-1]$ en el que
 - $T[i] = x$ si $x \in T$ y $\text{clave}[x] = i$
 - $T[i] = \text{NULL}$ en caso contrario
 - Es una *tabla de acceso directo*
 - Las operaciones son $O(1)$, la tabla de tamaño m !!!!
 - Qué pasa si insertamos 10000 DNIs?
 - ◆ Necesitamos una tabla de tamaño 100.000.000!!!

Problemas con el Acceso Directo

- Funciona correctamente cuando el rango de valores (m) para la clave es relativamente pequeño
- Que ocurriría si tenemos valores enteros de 32 bits?
 - Problema 1: La tabla de acceso directo tendrá 2^{32} entradas, más de 1 billon !!!
 - Problema 2: Incluso cuando tengamos memoria suficiente, el tiempo necesario para inicializar los elementos a NULL es muy costoso
- Solucion:
 - Utilizar una funcion que convierta los valores en el rango $0..m-1$
 - Esta funcion se la denomina función *hash*

Funciones Hash

- Cuando existe una función hash biyectiva hablamos de **función hash perfecta**: El conjunto de datos debe ser fijo y predeterminado.
- En el resto de los casos tendremos funciones sobreyectivas: para dos claves distintas podremos obtener un mismo valor. Se producen **colisiones** en el valor de la función **hash**.

Una función hash perfecta es difícil de encontrar, pero como podemos imaginar las podemos encontrar buenas funciones hash.....

Elección de la función Hash

- Criterios:

- Produzca el menor número de colisiones: Distribuir uniformemente las claves por la tabla
- Dependencia de todos los valores de la clave: Dos claves que se diferencien en un pequeño grupo de bits deben tener valores hash distintos.
- Debe ser sensible a permutaciones de los valores: 12345 y 24351 deben dar valores hash distintos
- Rápida y fácil de calcular

Funciones Hash en STL

Definidas en el archivo cabecera <functional>

```
template< class Key > struct hash;
```

```
...
```

```
int main()  
{
```

```
    string str = "Esto es una cadena...";
```

```
    hash< string> hash_fn;
```

```
    size_t str_hash = hash_fn(str);
```

```
    cout << str_hash << '\n';
```

```
}
```

Salida : 15500687590072717837



C++11

hash<key> en STL

Define un objeto función que implementa una función hash. Las instancias de esta función definen el operador () que:

1. Acepta un único parámetro de tipo Key.
2. Devuelve un valor de tipo size_t que representa el valor hash del parámetro.
3. No produce excepciones cuando se le llama.
4. Dados dos parámetros k1 y k2 que son iguales, hash <Key> () (k1) == hash <Key> () (k2).
5. Dados k1 y k2 distintos, la probabilidad de que hash <Key> () (k1) == hash <Key> () (k2) debe ser muy pequeña, 1.0/max valor de size_t

Tablas Hash

- Dadas las claves:

■ 1234567

■ 8765424

■ 7485731

■ 8457566

■ 1287398

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

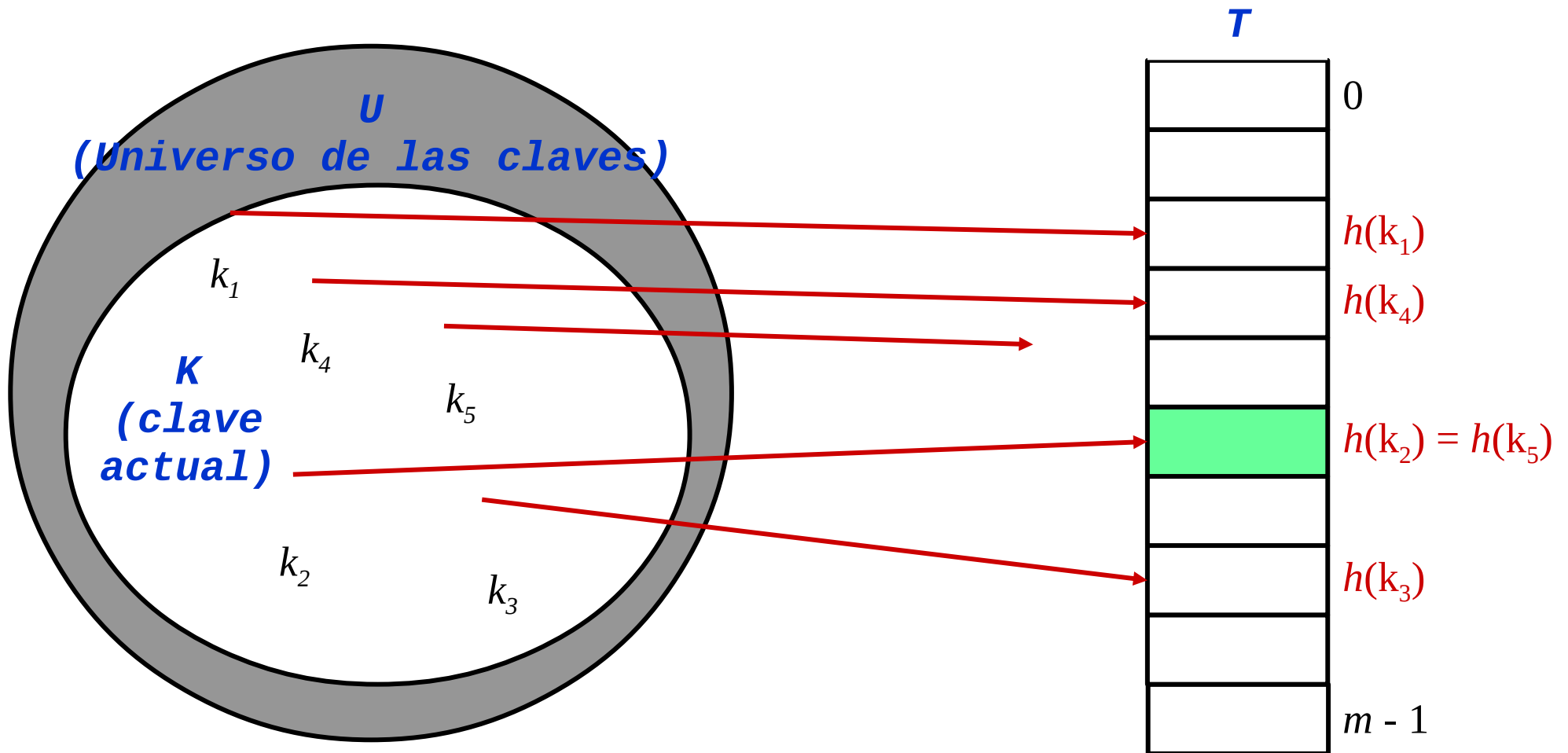
Funcion Hash: $h(k) = k \bmod 10$

Cómo se realiza la Insercion?

Cómo se realiza el Borrado?

Cómo se realiza la Busqueda?

Tablas Hash: Colisiones



Resolucion de colisiones

- Solucion 1: *Dispersión cerrada,*
 - *Idea: Insertar todas las claves en la tabla buscándoles una nueva ubicación tras una colisión*
 - *Válido si tenemos una estimación del numero de elementos a colocar en la tabla y no hay problema de memoria.*
 - *Hashing lineal: Encadenamiento*
 - *Hashing doble: Utilizar otra funcion hash.*
- Solucion 2: *Encadenamiento Separado*
 - *Idea: En cada posicion de la tabla, construir una lista enlazada con los elementos que caigan en la misma posicion*

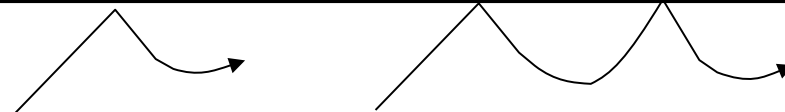
Disposición Cerrada

- Las celdas de la tabla contienen:
 - Elemento propiamente dicho
 - Key -> unordered_set
 - pair<Key,Def> -> unordered_map
 - Estado: Bandera que toma tres valores:
 - Libre:
 - Ocupada:
 - Borrada:

Dispersión cerrada:

- En caso de **colisión** es necesario hacer **redispersión**: buscar una nueva posición para meter el elemento.
- **Redispersión**:
 - si falla $h(k)$, aplicar $h_1(k)$, $h_2(k)$, ... hasta encontrar una posición donde podemos insertar (en el ejemplo buscar el sig. Hueco).
- Sea $M = 10$, $D = \{9, 33, 976, 285, 541, 543, 25, 2180\}$
 $h(k) = k \% M$

0	1	2	3	4	5	6	7	8	9
2180	541		33	543	285	976	25		9



La secuencia de posiciones recorridas para un elemento se suele denominar cadena o **secuencia de búsqueda**.

Dispersión Cerrada: Búsqueda de un elemento


Examinar la posición $h(k)$.

- Si x está en la posición $h(k)$ y estado = ocupada,
 - entonces devolver true (o la definición).
- Si estado= libre, entonces x no es miembro;
 - devolver False.
- En otro caso, la posición está ocupada pero por otro elemento. Debemos examinar otras posiciones $h_1(k)$, $h_2(k)$, ... y así sucesivamente hasta encontrar k , o encontrar la primera posición libre o examinar toda la tabla.

Dispersión cerrada: Borrado

- Buscar+Eliminar?
- **Ojo** con la eliminación.
- Al eliminar un elemento se puede destruir la secuencia de pasos necesaria.
 - Si k_i precede a k_j en una secuencia de pruebas, no podemos eliminarlo sin mas. Cuando queramos encontrar k_j alcanzamos un hueco y se determina que k_j no se encuentra en la tabla.
- **Ejemplo:** eliminar 976 y luego consultar 285.

285



0	1	2	3	4	5	6	7	8	9
2180	541		33	543	25		285		9

Resultado: ¡¡285 no está en la tabla!!

Dispersión cerrada

- **Conclusión:** en el borrado no se pueden romper las secuencias de búsqueda
- Solucion: Utilizar una etiqueta para marcar las celdas borradas
- Cada posicion puede estar : vacia, ocupada y borrada.
 - **Busqueda: ocupada == borrada**
 - **Insercion: borrada == vacia**

Dispersión cerrada: Resolución de colisiones lineal

- Idea: Inspeccionar una sucesión de localizaciones en la tabla

- Hashing: $h_1(k) = k \bmod M$
- tras colision $h_2(k) = (h_1(k) + 1) \bmod M,$
- $h_3(k) = (h_1(k) + 2) \bmod M,$

En general

- $h_i(k) = (h_1(k) + (i-1)) \bmod M$

Hashing lineal: $h(k) = k \bmod 13$

	K	$h_1(k)$
1	119	2
2	85	7
3	43	4
4	141	11
5	73	7
6	91	0
7	109	5
8	147	4
9	38	12
10	137	7
11	148	5
12	101	10

0	91	1
1		
2	119	1
3		
4	43	1
5	109	1
6	147	3
7	85	1
8	73	2
9		
10		
11	141	1
12	38	1

A
G
R
U
P
A
C
I
O
N

Hashing lineal: $h(k) = k \bmod 13$

	K	$h_1(k)$
1	119	2
2	85	7
3	43	4
4	141	11
5	73	7
6	91	0
7	109	5
8	147	4
9	38	12
10	137	7
11	148	5
12	101	10

0	91	1
1	101	5
2	119	1
3		
4	43	1
5	109	1
6	147	3
7	85	1
8	73	2
9	137	3
10	148	6
11	141	1
12	38	1

Problema: Agrupaciones

- Consiste en el aumento de la probabilidad de una colision cada vez que insertamos los elementos.
 - La probabilidad de que la primera insercion ocupe la posicion L1, $P(i_1=L1) = 1/13$
 - $P(i_2=L2) = 2/13$, tanto L1 como L2 ocupan la posicion L2
 - $P(i_3=L3) = 3/13, \dots$
- Problema!: Claves con el mismo valor hash utilizaran la misma secuencia para la resolucio[n] de colisiones.
- Solucion: Evitar este comportamiento:

Hashing doble

Dispersión cerrada: Hashing doble

- Utiliza dos funciones hash
 - Inicial: $h_1(k) = k \bmod M$
 - Si hay colisión, entonces emplea
$$h_i(k) = (h_{i-1}(k) + h_0(k)) \bmod M, \text{ con } i=2,3,\dots$$
$$h_0(k) = 1 + (k \bmod (M-2))$$

Funciona bien cuando:

$$h_0(k) \neq 0$$

M y $M-2$ son primos relativos

Garantiza el acceso a todas las posiciones

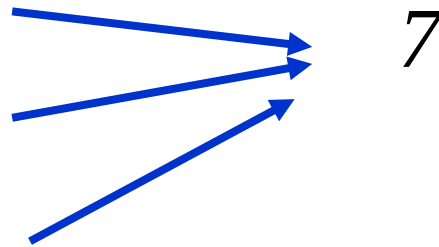
Hashing doble:Ejemplo

- $h_1(k) = k \bmod 13$

- $k_0 = 7$

- $k_1 = 85$

- $k_2 = 72$



$$\begin{aligned} h_2(k_1) &= (h_1(k_1) + h_0(k_1)) \bmod 13 = \\ &= (7 + (1 + k_1 \bmod 11)) = (7 + 9) \bmod 13 = 3 \end{aligned}$$

$$\begin{aligned} h_2(k_2) &= (h_1(k_2) + h_0(k_2)) \bmod 13 = \\ &= (7 + (1 + k_2 \bmod 11)) = (7 + 7) \bmod 13 = 1 \end{aligned}$$

Ejemplo: $h(k) = k \bmod 13$, $h_0(k) = 1 + k \bmod 11$

	K	$h_1(k)$	$h_0(k)$
1	119	2	10
2	85	7	9
3	43	4	11
4	141	11	10
5	73	7	7
6	91	0	4
7	109	5	11
8	147	4	5
9	38	12	6
10	137	7	6
11	148	5	6
12	101	10	3

0	91	1
1	72	2
2	119	1
3	101	3
4	43	1
5	109	1
6	137	3
7	85	1
8		
9	147	2
10	148	4
11	141	1
12	38	1

Dispersión cerrada: Eficiencia

La tabla nunca se puede llenar con más de M elementos.

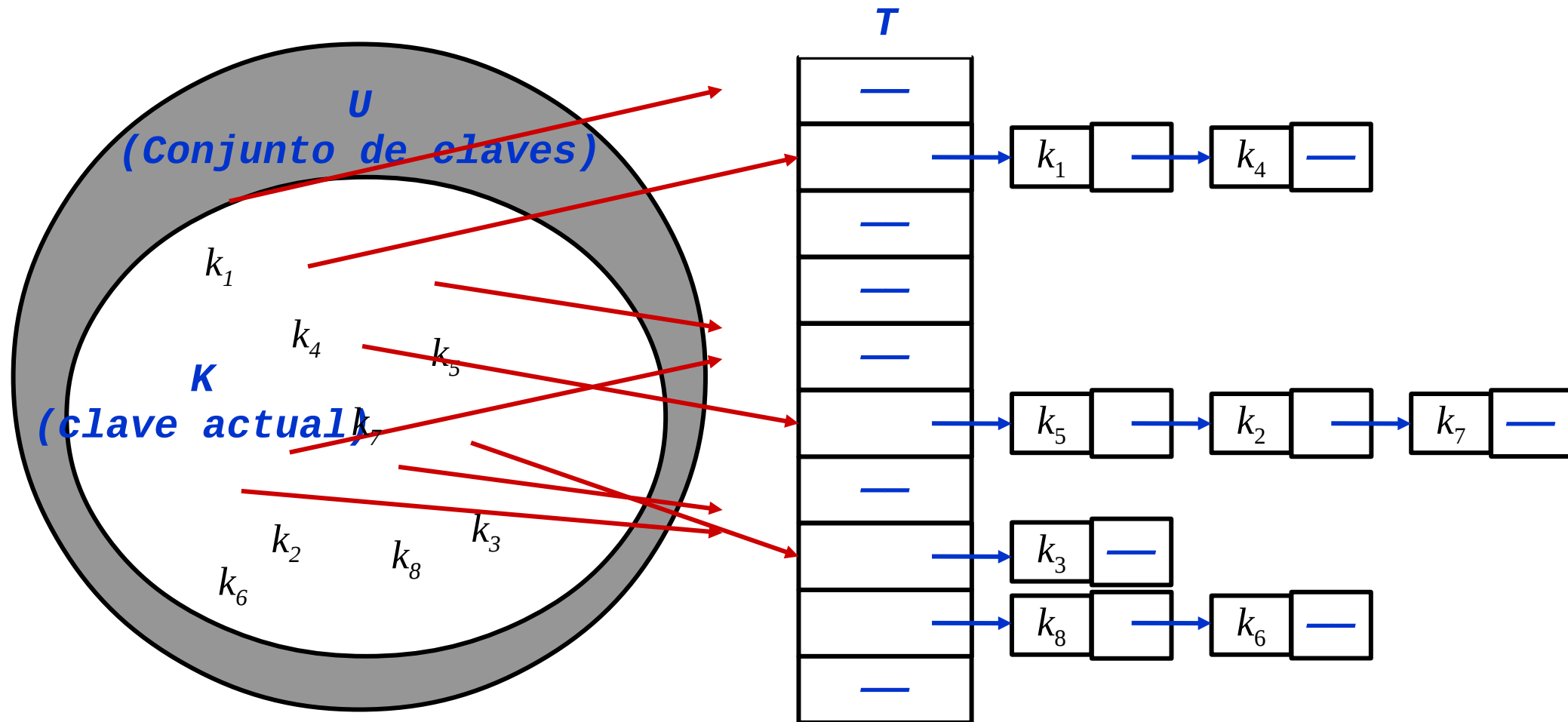
La probabilidad de colisión crece cuantos más elementos hayan, disminuyendo la eficiencia.

El costo de Insertar es $O(1/(1-n/M))$

Cuando $n \rightarrow M$, el tiempo tiende a infinito.

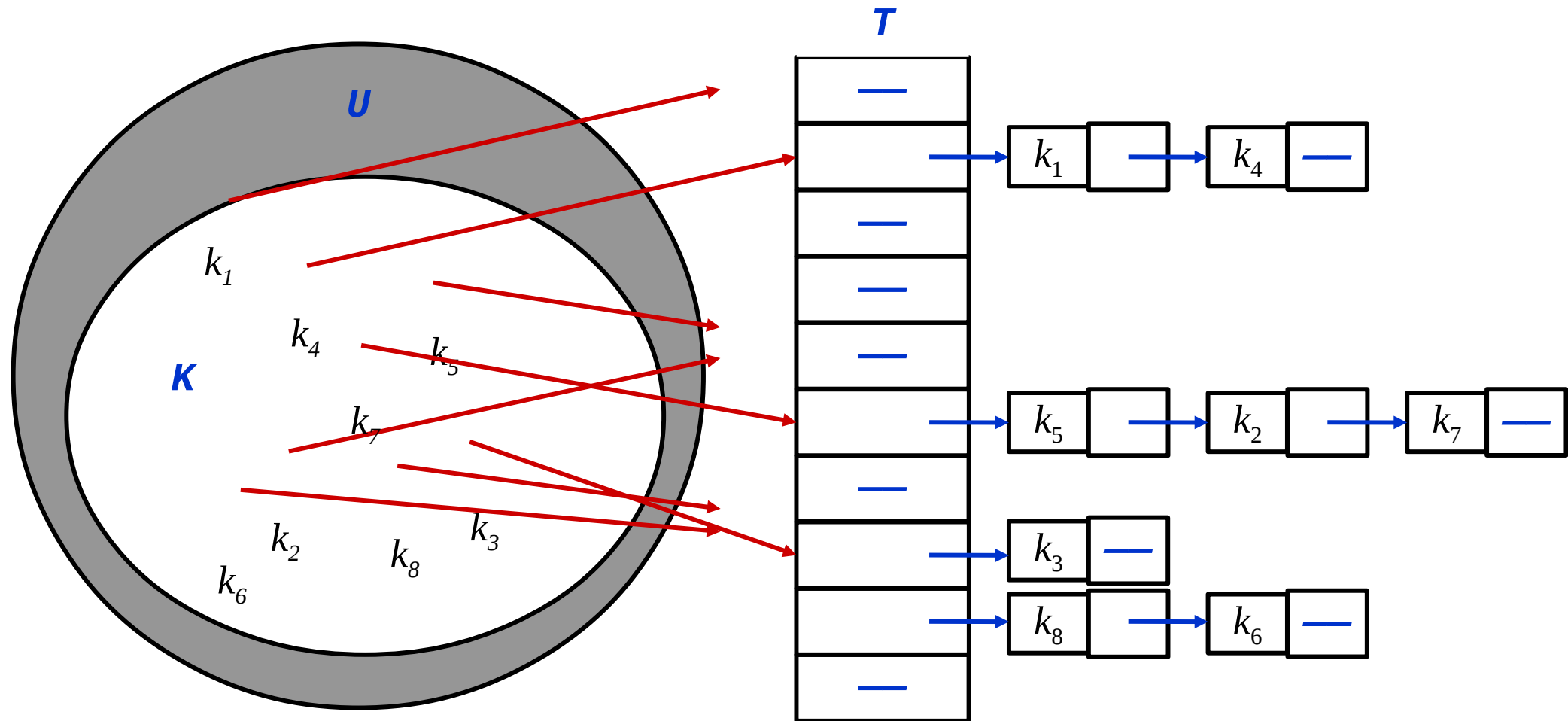
Encadenamiento separado

- Encadena los elementos con una misma funcion hash en una lista enlazada



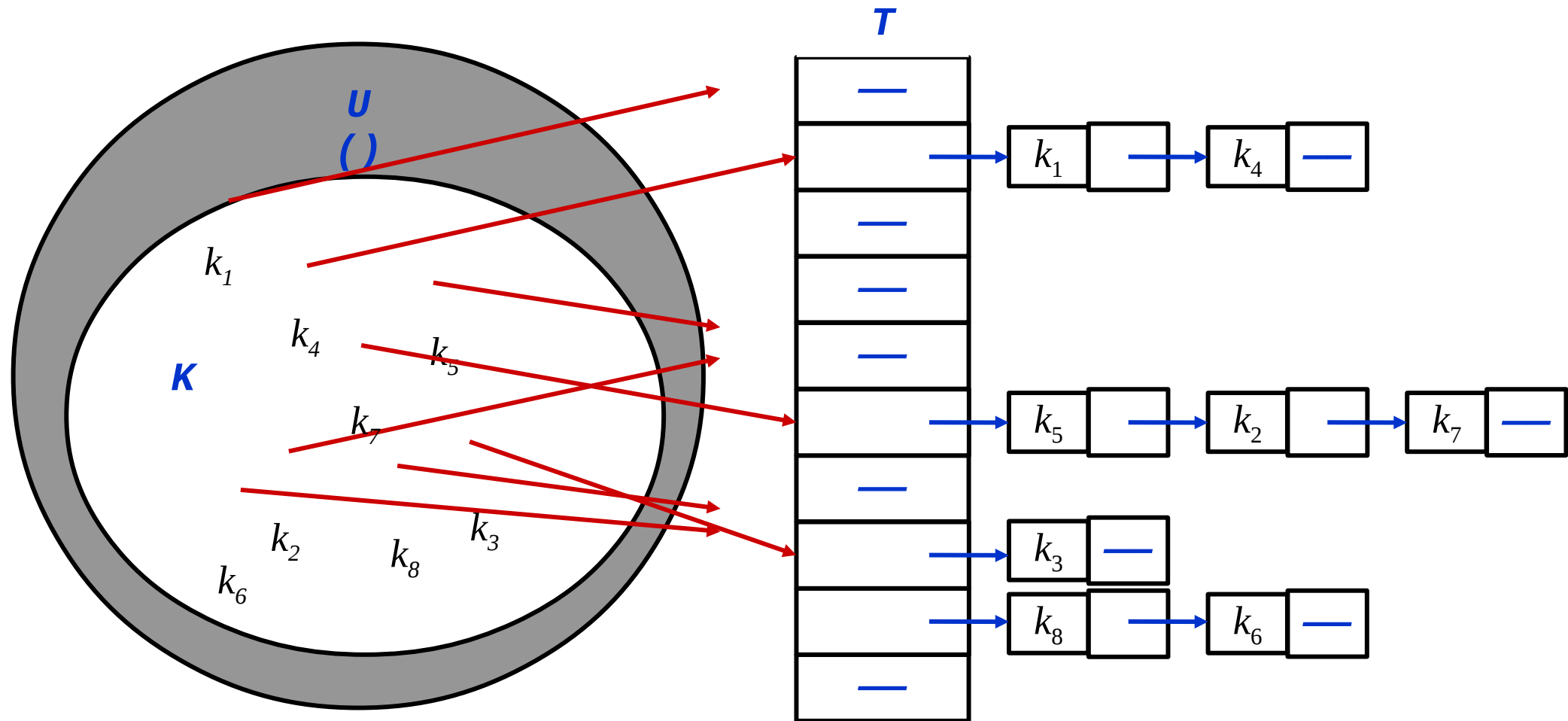
Encadenamiento separado

- *Como insertamos un elemento?*



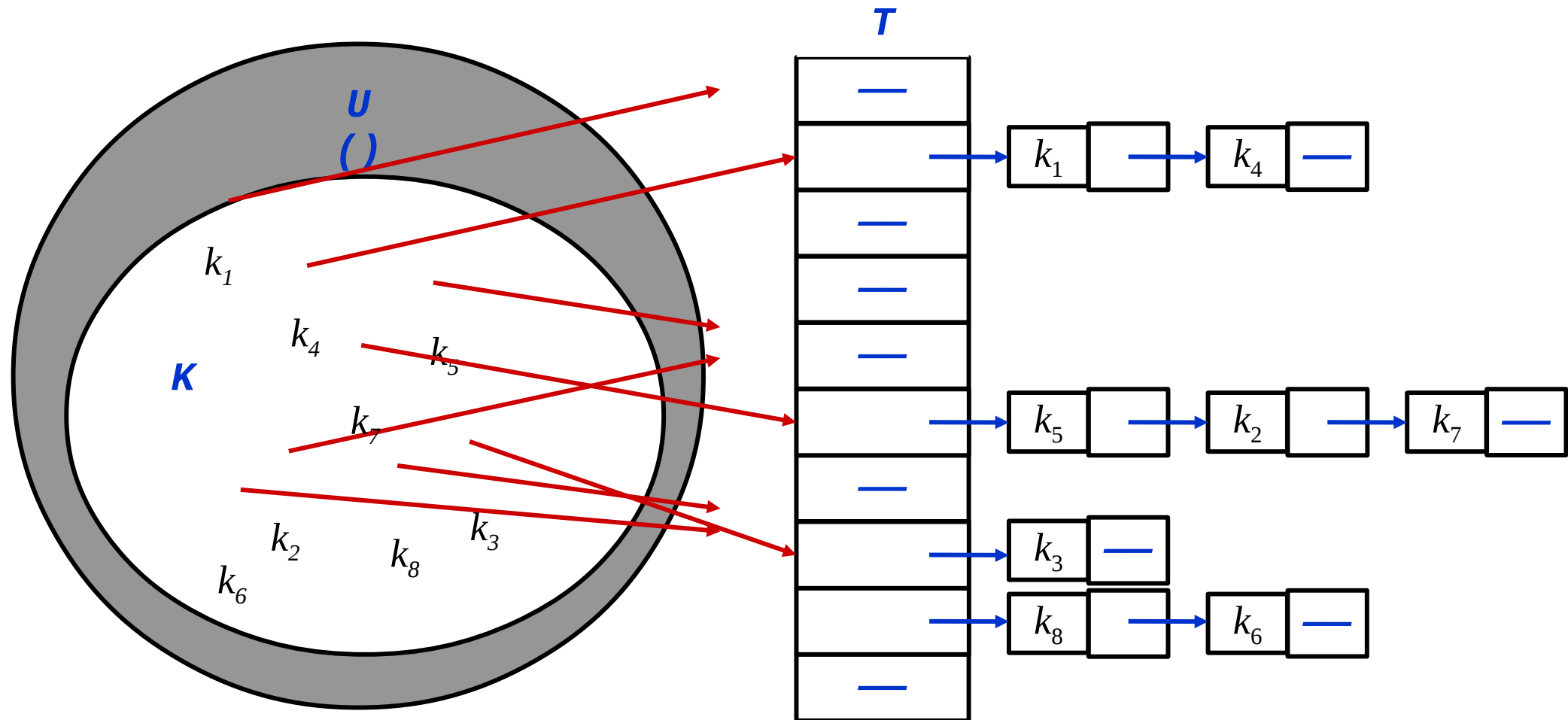
Encadenamiento separado

- *Como borramos un elemento?*



Encadenamiento separado

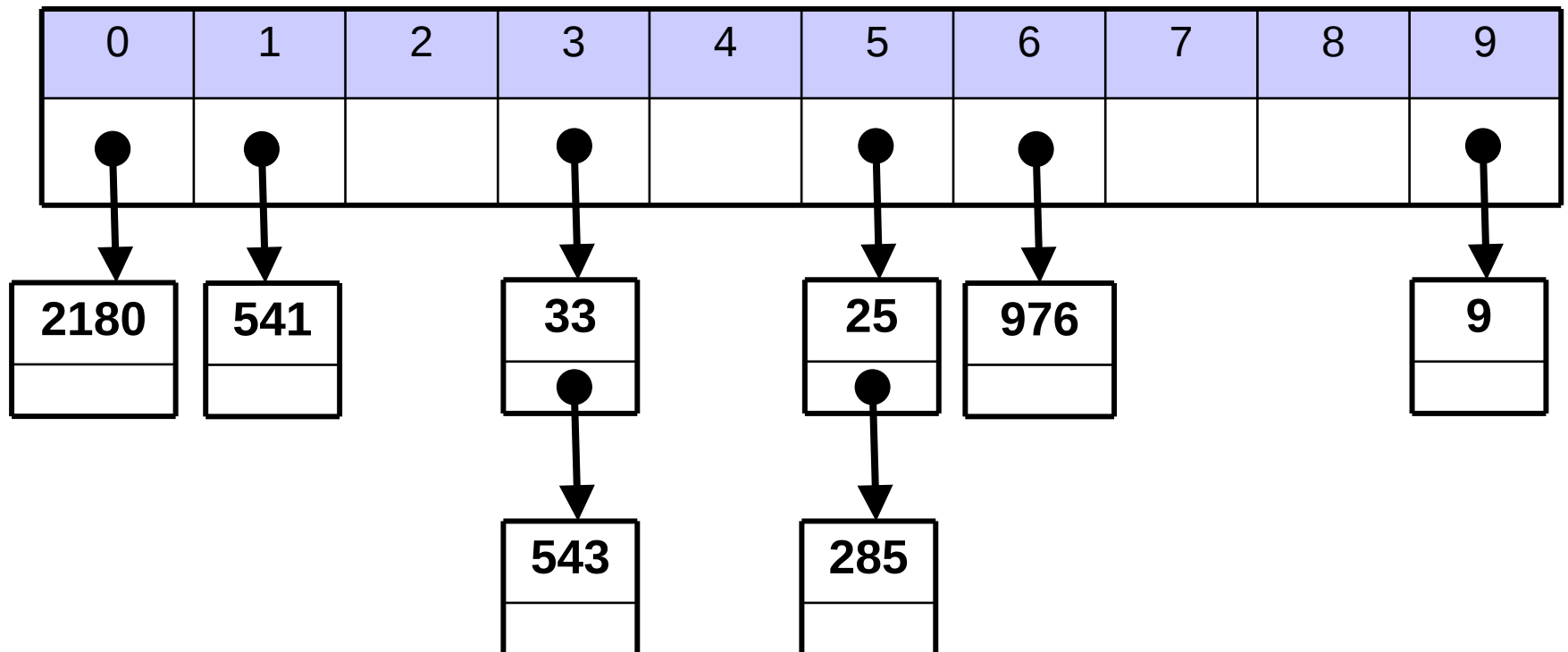
- *Como buscamos un elemento?*



Ejemplo

- Sea $M = 10$, $D = \{9, 25, 33, 976, 285, 541, 543, 2180\}$
- $h(k) = k \% M$

T:



Analisis del Encadenamiento separado

- Asumimos una funcion hash uniforme: Todas las claves tienen la misma probabilidad de ser enviadas a cualquier casilla
- Dada n claves y m casillas en la tabla: el *factor de almacenamiento* $\alpha = n/m =$ media de claves por casilla
- *Cual será el costo de una busqueda fallida?*
 - R: $O(1+\alpha)$
- *Cual sera el costo medio de una busqueda con exito?*
 - R: $O(1 + \alpha/2) = O(1 + \alpha)$

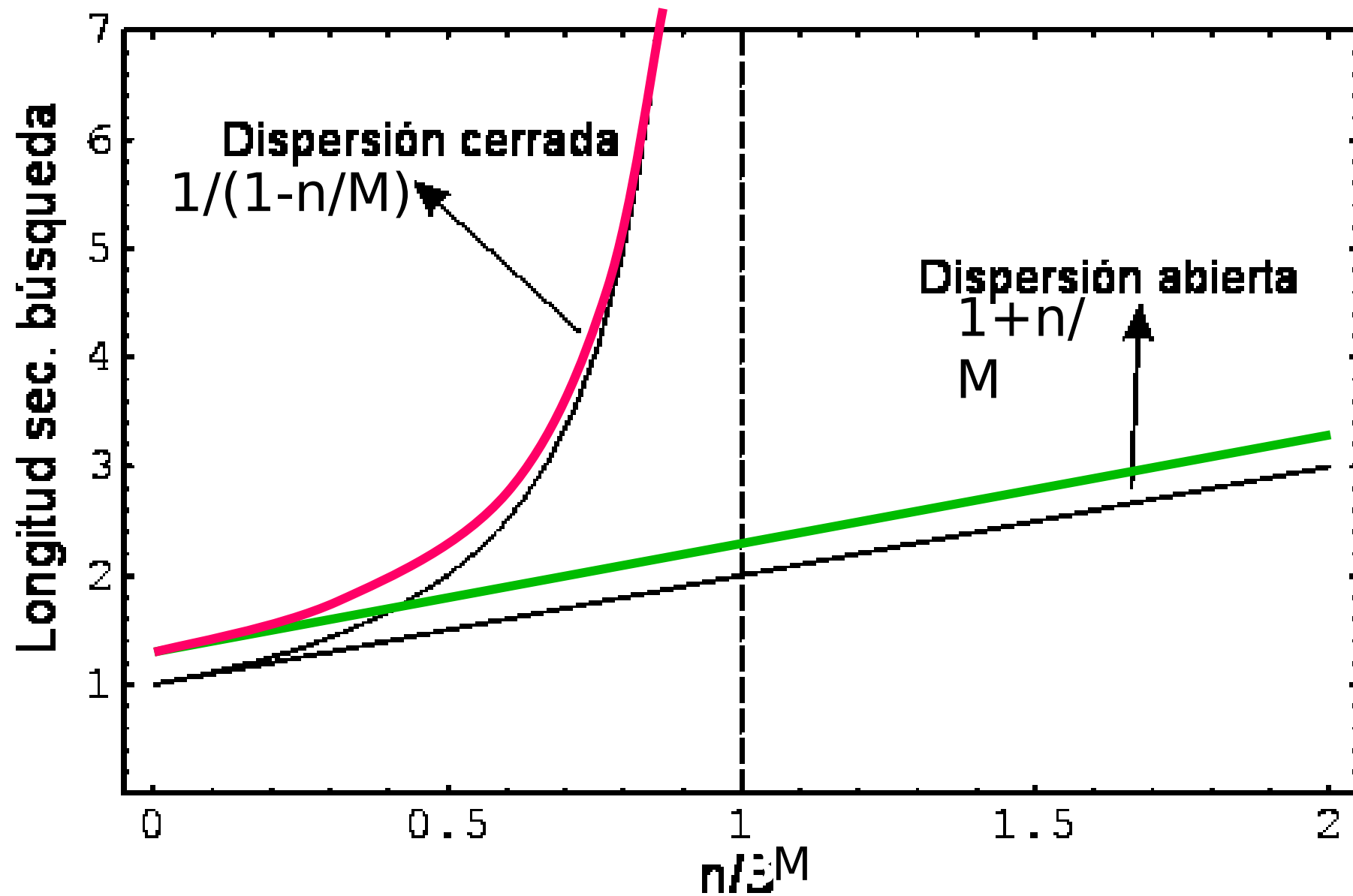
Análisis del Hashing abierto

- El costo de buscar es $= O(1 + \alpha)$
- *Si el numero de claves es proporcional al numero de casillas en la tabla, α , cual es el tiempo empleado en buscar un elemento?*
- *R: $\alpha = O(1)$*
 - En otras palabra, podemos esperar que el costo de buscar un elemento en la tabla sea constante si hacemos que α sea constante.

Análisis del Hashing abierto (cont)

- **Utilización de memoria**
- Si 1 puntero = p_1 bytes, 1 elemento = p_2 bytes.
- En las celdas: $(p_1 + p_2)n$
- En la tabla: $p_1 \cdot M$
- **Conclusión:**
- **Menos cubetas:** se gasta menos memoria.
- **Más cubetas:** operaciones más rápidas.

Comparativa



Eleccion de una Funcion Hash


- La eleccion correcta de una funcion hash es esencial para el buen funcionamiento de las tablas hash
- *Que características son deseables?*
 - - Produzca el menor número de colisiones: Distribuir uniformemente las claves por la tabla
 - - Dependá de todos los valores de la clave: Dos claves que se diferencien en un pequeño grupo de bits deben tener valores hash distintos.
 - - Debe ser sensible a permutaciones de los valores: 12345 y 24351 deben dar valores hash distintos
 - - Rápida y fácil de calcular

Funciones Hash:

El método de Division

- $h(k) = k \bmod m$
 - la casilla destino se obtiene como el resto de dividir k entre m , donde m representa el tamaño de la tabla
- *Que ocurre con elementos con claves consecutivas?*
 - *R: Ocupan distintas posiciones, pero tienden a formar agrupaciones*
- *Que ocurre si m es potencia de 2 ($m = 2^p$)?*
 - *R: Solo influyen los ultimos p bits de la clave*
- *Que ocurre si m es potencia de 10?*
 - *R: Igual, solo influyen los ultimos digitos de la clave*
- Solucion, hacer que $m =$ numero primo no muy cercano a una potencia de 2 (o 10)

Funcion Hash: Metodo multiplicativo

- Dada una constante A , $0 < A < 1$:
- $h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$  *Parte entera*

- Escoger $m = 2^P$

- Escoger A no muy cercano a 0 o 1

- Knuth propone $A = (\sqrt{5} - 1)/2 = 0.61803399$

- Ejemplo: $m = 128$, $k = 567$

- $h(k) = \lfloor 128 * (350.42527 - 350) \rfloor = \lfloor 128 * 0.42527 \rfloor =$

$h(k) = \lfloor 54.434 \rfloor = 54$

Otras funciones Hash:

- Método de multiplicación y División

$$h(x) = (ax + b) \bmod M$$

con $(a \bmod M) \neq 0$

Dependiendo del tipo de dato que sea la clave:

- Si es un entero podemos utilizar como función una de las vistas anteriormente, por ejemplo $h(x) = x \bmod M$
- Si la clave es un string

$$h(x) = (x[0] + \dots + x[n-1]) \bmod M$$

con n la longitud de x

Rehashing

- Problema:
 - Una tabla hash llega al desbordamiento: El numero de elementos en la tabla hash es muy cercano a M
 - La eficiencia ha disminuido por tener muchos borrados.
- Solucion:
 - Llevar los elementos a una nueva tabla (incluso aumentando su tamaño si es necesario)

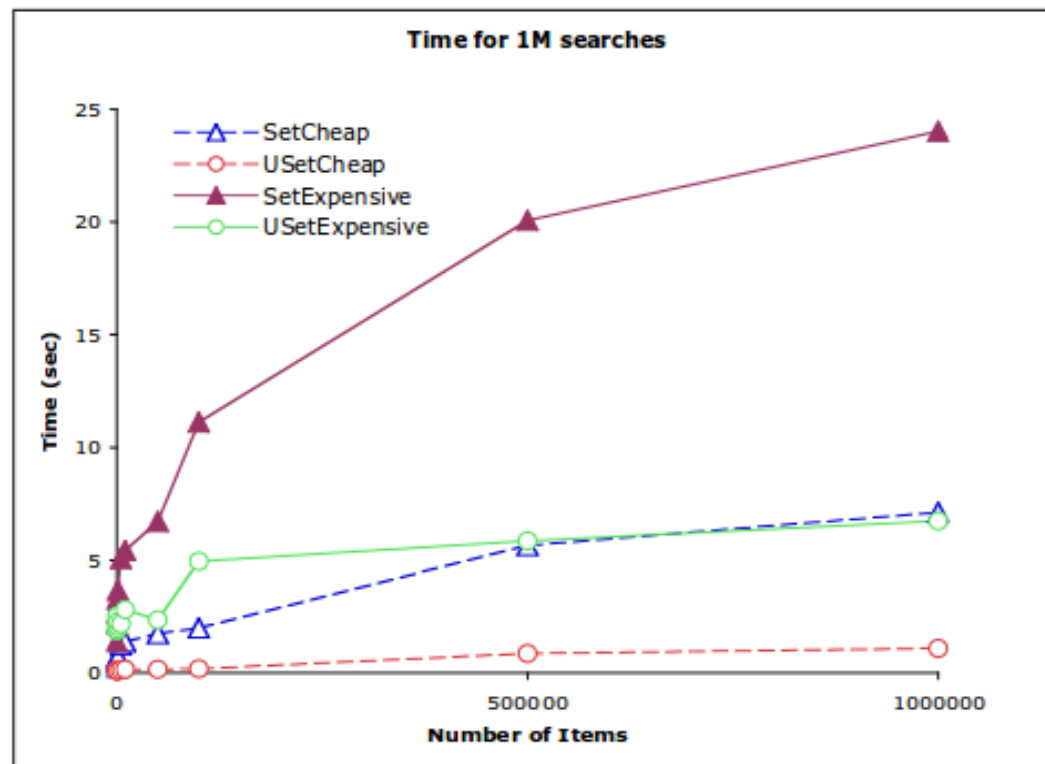
Algunas notas sobre la eficiencia

- **BE**: busquedas con Éxito, **BF**: busq. con Fracaso
- **FA**: factor almacenamiento = $n_elementos / m$

	H. Separado		H. Lineal		H. Doble	
FA	BE	BF	BE	BF	BE	BF
0.25	1.12	1.03	1.17	1.39	1.15	1.33
0.50	1.25	1.11	1.50	2.50	1.39	2.00
0.75	1.38	1.22	2.50	8.50	1.85	4.00
0.90	1.45	1.31	5.50	50.5	2.56	10.0
0.95	1.58	1.43	10.5	200	3.15	20.0

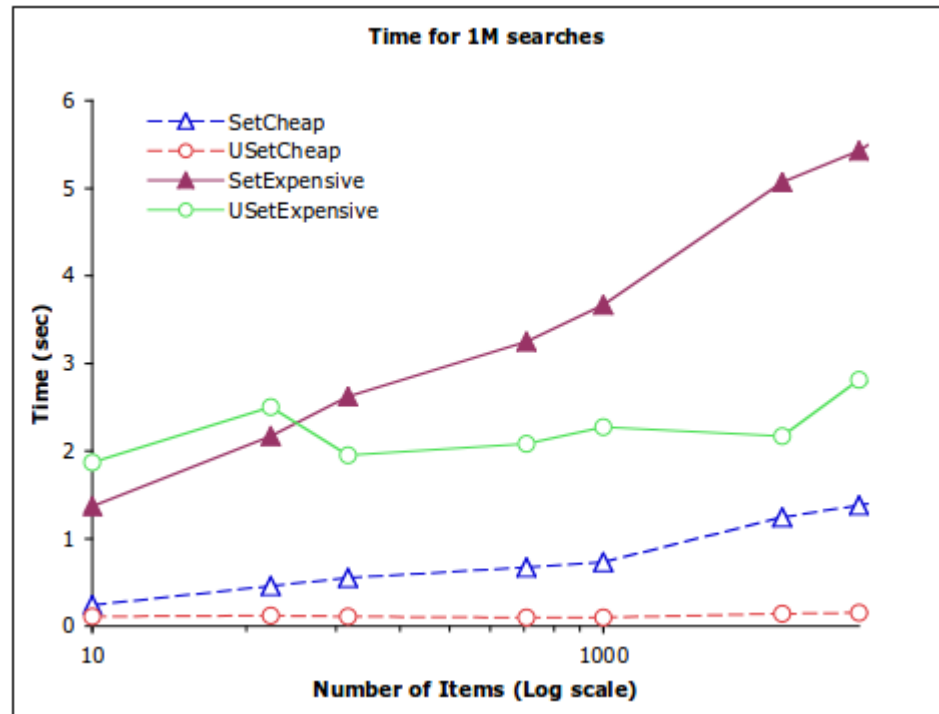
Experimento: comparar set-unorder

- Método find:
- Elementos: Cheap (int) Expensive (cadenas largas con igual prefijo)
- Tamaño contenedor
{10, 50, 100, 500, 1000, 5000, 10,000, 50,000, 100,000, 500,000, 1,000,000}



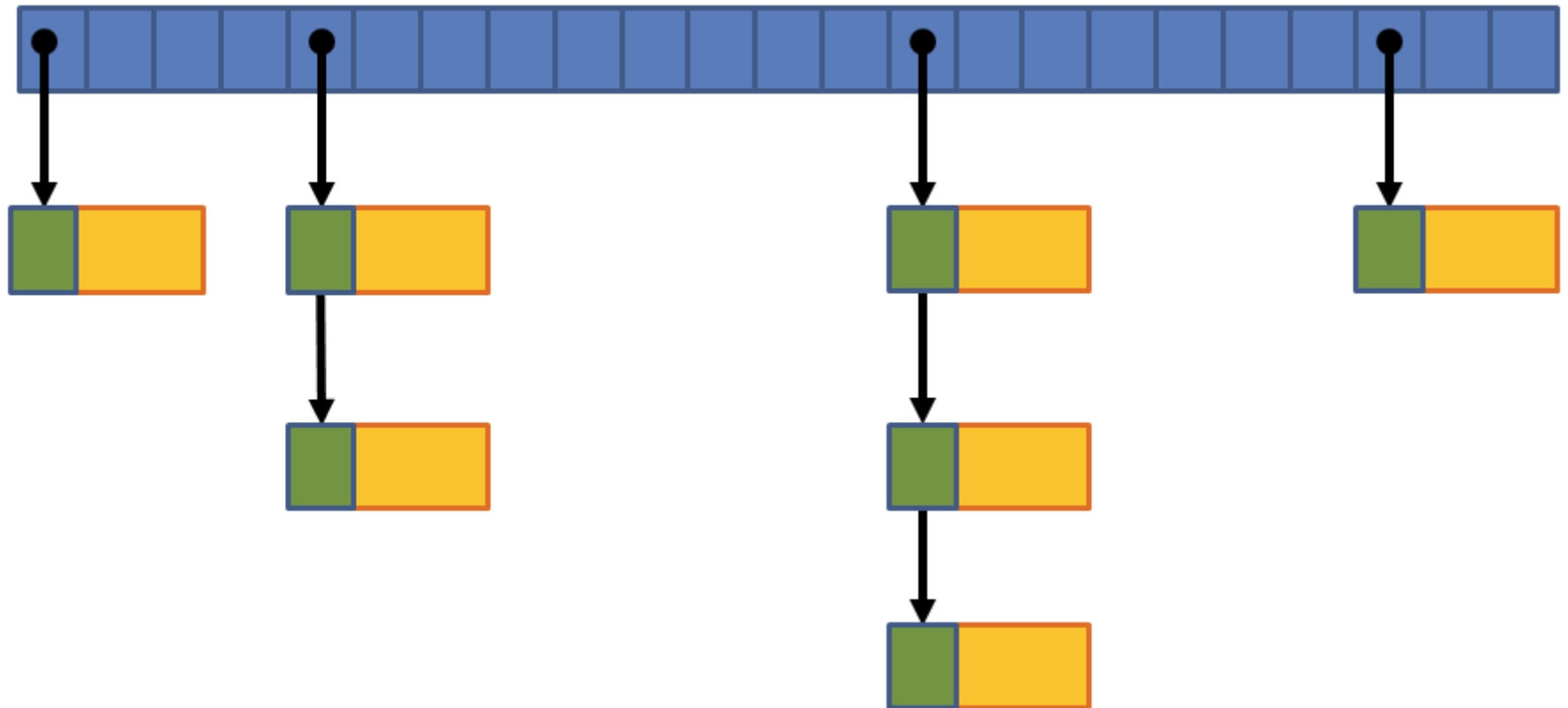
Comparar set-unordered set

- Zoom sobre los valores pequeños



STL y Tablas Hash

- `unordered_map<Key,Def>` : implementa hashing por encadenamiento separado



unordered_map:

```
template < class Key,  
          class T,  
          class Hash = hash<Key>,  
          class Pred = equal_to<Key>, // para colis.  
          class Alloc = allocator< pair<const Key,T> >  
        > class unordered_map;
```

.....

```
unordered_map<string, int> UM;
```

```
unordered_map<string, int> indice;  
// equivalente a  
unordered_map<string, int, hash<string>, equal_to<string> > indice;
```

```
classs PersonaHash  
{ public:  
    size_t operator()(const Persona& p) const  
    {  
        return hash<string>()(p.Name());  
    }  
};
```

// En caso de colisiones

```
class PersonaIgualdad  
{  
    bool operator()(const Persona& l, const Persona& r) const  
    {  
        return l.Name() == r.Name();  
    }  
};  
unordered_map<Persona, int, PersonaHash, PersonaIgualdad> UmPers;
```

Unordered_map: Funciones Miembro

Capacidad

- `empty` : Chequea si está vacío
- `size`: Número de elementos en el contenedor
- `max_size`: Máximo número de elementos

Iteradores:

- `Begin()`; Iterador al primer elemento del contenedor
- `End()`; Iterador al siguiente al último elemento

Acceso a los elementos:

- `at()` acceso por clave (si no está, lanza excepción) $O(1)$
- `find` Devuelve iterador al elemento (si no se encuentra `end`) ($O(1)$)
- `count` Cuenta el número de elementos con un valor de clave $O(1)$

Unordered_map: Funciones Miembro

Modificadores

- `insert` : Inserta un elemento en el map, el elemento es de tipo `value_tipo` esto es un `pair<Key,Def>`. El elemento no puede estar en el map, si está la insercion no se puede hacer, pues las claves son únicas. $O(1)$

Devuelve `pair<iterator,bool>`

- `Operator[]` acceso por clave , devuelve la referencia al la Def del elemento con clave K, si existe. Si igual que map, Inserta si no está inserta el elemento asignándole como valor `Def()`, $O(1)$

Ejemplo

```
unordered_map<string,int> UM;  
UM["Pedro"]=123;  UM["Luis"] = 3242;  
UM["Sara"]=32;  UM["Ana"] = 324;  
UM["Juan"] = 2;  UM["Rosa"]= 4;  
UM.insert(pair<string,int>("Marta",23));  
    unordered_map<string,int>::iterator it;  
for (it = UM.begin();it!=UM.end();++it){  
    cout << (*it).first<< " " ;  
}  
if (Um.find("Ruben") == UM.end()) cout << "NO está·";  
else (*it).second = 2435;
```

Salida: Ana Luis Pedro Rosa Juan Sara Marta

unordered_map<Key,Def>: Control de la tabla hash.

- Como hemos visto, se adopta un interfaz como la del map, pero permite tener algunos métodos para controlar la tabla hash:

Conceptos importantes:

- [Bucket](#):
- [load_factor](#):

unordered_map<Key,Def>

- **Bucket**: Un bucket (cubo) es una posición dentro del contenedor que almacena los elementos de la tabla hash (celda del vector).
 - Los elementos se asignan a la celda en función de su valor hash.
 - Numerados desde cero hasta bucket_count-1.
 - Por defecto, se crean 11 buckets
- **load_factor**: Factor de almacenamiento, definido como el ratio entre el numero de elementos en la tabla hash y el numero de buckets
$$(\text{load_factor} = \text{size}/\text{bucket_count}).$$
 - El load_factor influye en la probabilidad de colisiones en la tabla hash,
 - El contenedor aumenta el número de buckets para asegurar que el factor de almacenamiento se encuentra bajo un umbral ==> rehash.

Unordered_map<Key,Def>: Funciones Miembro

Buckets

- `bucket_count()` numero de buckets
- `max_bucket_count()` máximo número de buckets
- `bucket_size(int i)` Numero de elementos en el buckets i-esimo0
- `bucket(key k)` Devuelve el numero del bucket para el elemento k
- `local_iterator begin(int i)` Iterador a los elementos del bucket i-ésimo
- `local_iterator end(int i)` fin de los elementos del bucket i-esimo

Ejemplo de uso

```
n = UM.bucket_count() ;  
cout << "bucket count " << UM.bucket_count() << endl;  
  
for (unsigned i=0; i<n; ++i) {  
    std::cout << "bucket #" << i << " contiene: " << endl;  
    for ( unordered_map<string,int>::local_iterator it2 =  
          UM.begin(i); it2!=UM.end(i); ++it2)  
        cout << "[" << it2->first << ":" << it2->second << "] ";  
    cout << endl ;  
}
```

Unordered_map<Key,Def>: Funciones Miembro

Gestión del hash

- `load_factor()` Devuelve load factor
- `max_load_factor()`
- `max_load_factor(float z)`
 - Devuelve o asigna el máximo factor de almacenamiento, por defecto 1.0
- `rehash(size_type n)` Asigna el numero de buckets, n. Si n es > bucket_count, se reajusta para obtener un nuevo numero de búckets ($\geq n$, pues se busca un numero primo). Todos los elementos son movidos a la nueva tabla.
- `reserve(size_type n)` Hace una reserva para el numero buckets a n

```
unordered_map<int,int> Ui;  
size_t ant, desp;  
double fac_ant,fac_desp;  
  
for (int i=0;i<1000; i++){  
    ant = Ui.bucket_count(); fac_ant = Ui.load_factor();  
    Ui[i]=i;  
    desp = Ui.bucket_count(); fac = Ui.load_factor()  
    if (ant != desp)  
        cout<< i <<": " << ant << "-> " << desp << " " << fac << endl;  
}
```

Salida:

11: 11-> 23 0.521739

23: 23-> 47 0.510638

47: 47-> 97 0.494845

97: 97-> 199 0.492462

199: 199-> 409 0.488998

409: 409-> 823 0.498177

823: 823-> 1741 0.473291

```
unordered_map<int,int> Ui;  
size_t ant, desp;  
double fac_ant,fac_desp;  
Ui.max_load_factor(2.5);  
for (int i=0;i<1000; i++){  
    ant = Ui.bucket_count();  
    Ui[i]=i;  
    desp = Ui.bucket_count(); fac = Ui.load_factor()  
    if (ant != desp)  
        cout<< i <<": " << ant << "-> " << desp << " " << fac << endl;  
}
```

Salida:

```
28: 11-> 23 1.26087  
58: 23-> 47 1.25532  
118: 47-> 97 1.2268  
243: 97-> 199 1.22613  
498: 199-> 409 1.22005
```