

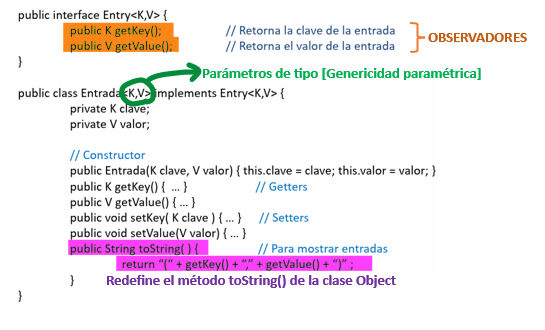
RESUMEN DE MAPEO Y DICCIONARIOS:

**ENTRADA:**

* **CLAVE:** Atributo de un individuo que sirve para identificarlo en un conjunto de individuos

**[EJEMPLO: DNI, Numero de libreta para alumnos, etc.]**

* Esto permite indexar con indices que no son enteros
* Para asociar calves con valores, utilize una **ENTRADA**

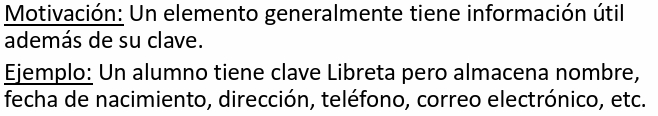


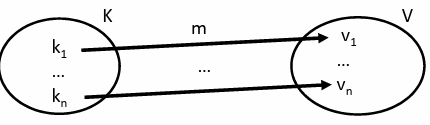
**implementación de Entrada**

**TDA MAPEOS:**

* **DEFINICION:** Es un conjunto de pares (K,V), donde:
* **K** 🡪 Es una clave
* **V** 🡪 Es el valor del elemento cuya clave es K

**Cada uno de los pares se define como una entrada en el mapeo e =(k,v)**

*  Un mapeo permite almacenar elementos de tal manera que puedan ser localizados rápidamente usando una clave
* Un mapeo puede ser interpretado como una **función (o correspondencia) de dominio K y codominio V**



* Un mapeo puede ser interpretado como un **conjunto de entradas** **m = {(ki,vi), i=1, … ,n}** donde ki tiene tipo K y vi tiene tipo V:



**Conjunto de pares [clave, valor][Pares ordenados] [Entradas]**



**Algunas cosas que podemos hacer:**

1. **Buscar clave y cambiar valor**
2. **Buscar un valor por su clave**
3. **Buscar clave y remover valor**

* Un **TDAMapeo** se modela a travez de una interfaz que se encuentra especificada a traves de las siguientes operaciones **[Dado un mapeo M]**:

1. **Size()** 🡪 Retorna el número de entradas de M
2. **isEmpty()** 🡪 Testea si M es vacío
3. **Get(k)** 🡪 Si M contiene una entrada e con clave igual a k, retorna el valor de v; sino retorna NULL **(\*)**
4. **Put(k,v):** **(\*)**

* Si M no tiene una entrada con clave k, entonces agrega una entrada (k,v) a M y retorna NULL
* Si M ya tiene una entrada e con clave k, reemplaza el valor con v en e y retorna el valor viejo de e

1. **Remove(k): (\*)**

* Remueve de M la entrada con clave k y retorna su valor
* Si M no tiene entrada con clave k, retorna NULL

1. **Keys():**

* Retorna una colección iterable de las claves en M
* **keys().iterator(): retorna un iterador de claves**

1. **Values():**

* Retorna una colección iterable con los valores de las claves almacenadas en M
* **values().iterator(): retorna un iterador de valores**

1. **Entries():**

* Retorna una colección iterable con las entradas de M
* **entries().iterator() retorna un iterador de entradas**

**(\*) InvalidKeyException: Excepcion de clave invalida [clave nula]**

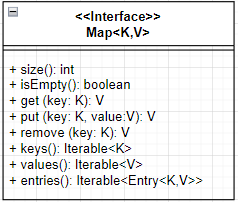
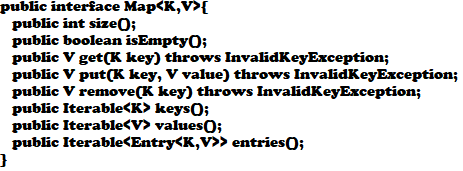
* **Un mapeo no puede almacenar claves repetidas [controlar esto cuando se inserta una nueva entrada en el mapeo]**

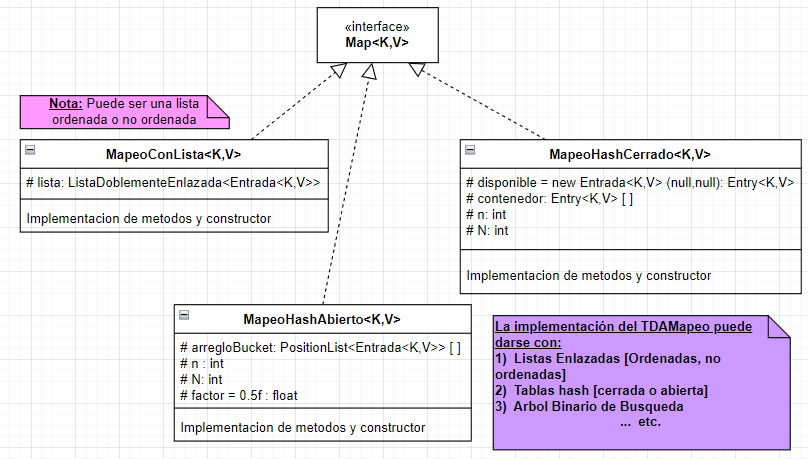
EJEMPLO DE LOS METODOS:



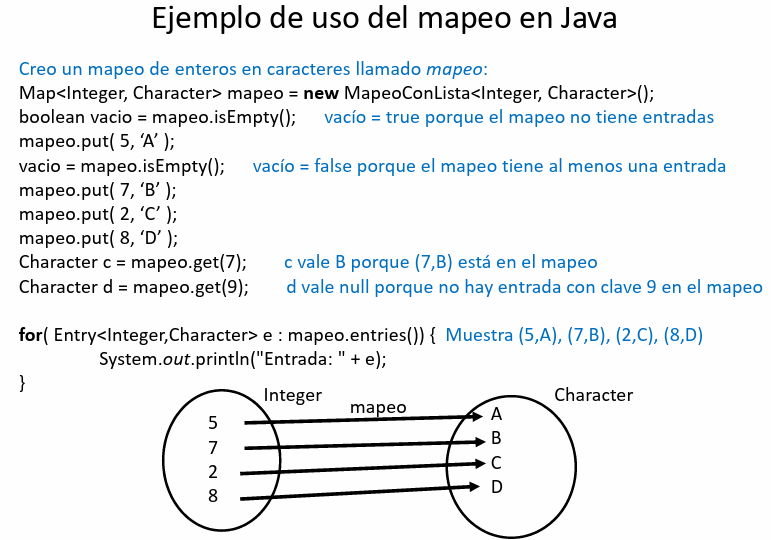
IMPLEMENTACION DE LA INTERFAZ MAPEO EN JAVA:

DIAGRAMA UML DE LA INTERFAZ MAPEO:





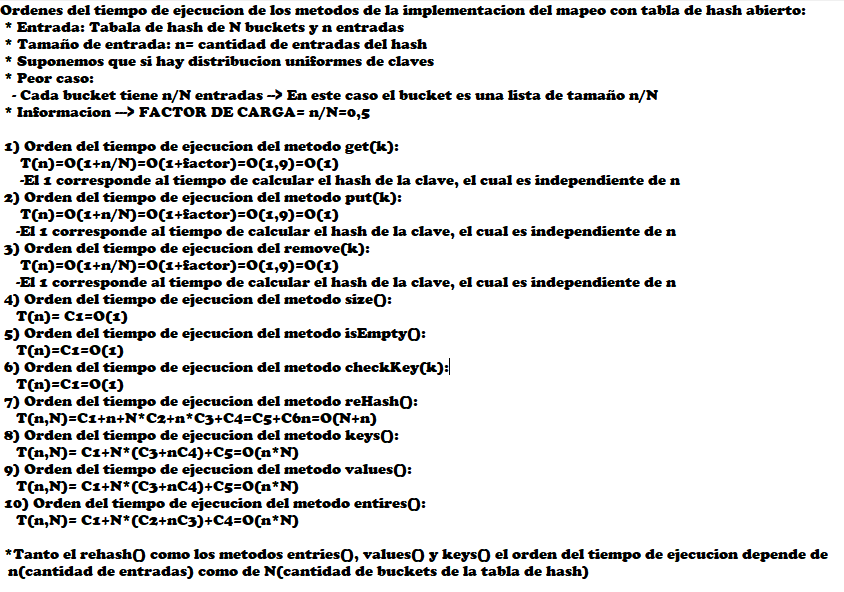
IMPLEMENTACIONES POSIBLES DE LA INTERFAZ MAP <K, V>:



**//Ver implementaciones en Java [MapeoConLista<K,V> , MapeoHashAbierto<K,V> y MapeoHashCerrado<K,V>]**

|  |  |  |  |
| --- | --- | --- | --- |
| **METODOS** | **ORDEN DEL TIEMPO DE EJECUCION** | | |
|  | **TIPO DE IMPLEMENTACION** | | |
|  | **Con Lista** | **Con hash Abierto** | **Con hash Cerrado** |
|  |  |  |  |
| **Size()** | **O(1)** | **O(1)** | **O(1)** |
| **isEmpty()** | **O(1)** | **O(1)** | **O(1)** |
| **Get(k)** | **O(n)** | **O(1)** | **O(n)** |
| **Put(k,v)** | **O(n)** | **O(1)** | **O(n)** |
| **Remove(k)** | **O(n)** | **O(1)** | **O(n)** |
| **Keys()** | **O(n)** | **O(n\*N) = O(n2)** | **O(n)** |
| **Values()** | **O(n)** | **O(n\*N) = O(n2)** | **O(n)** |
| **Entries()** | **O(n)** | **O(n\*N) = O(n2)** | **O(n)** |
| **rehash()** | **-** | **O(n+N)** | **-** |
| **hashThisKey(k)** | **-** | **O(1)** | **O(1)** |

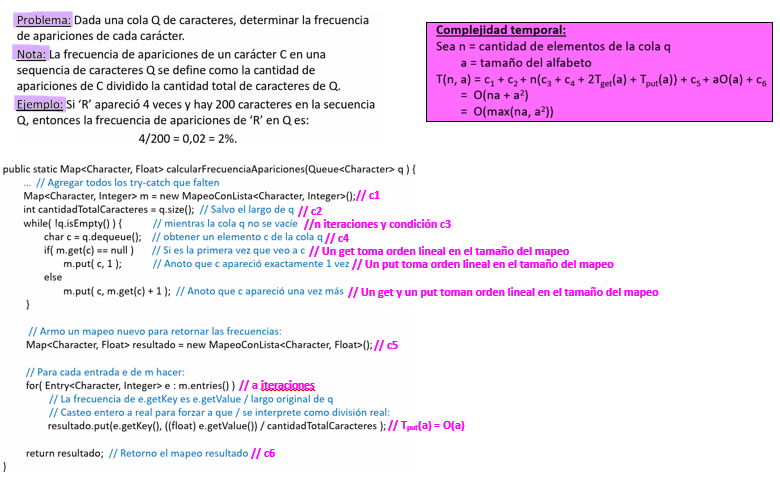
**¡No sé si es así!**

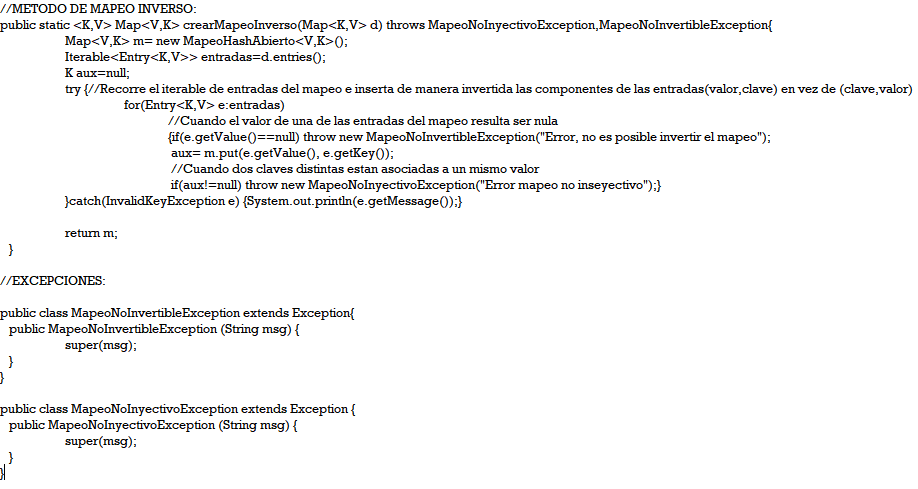




Explicación de orden del tiempo de ejecución de los métodos de la implementación con tabla de hash abierto:

EJEMPLO DE APLICACIÓN CON MAPEO [implementación con lista]:





EJEMPLO DE METODO PARA CREAR MAPEOS INVERSOS:



T(n) = O(n)



//Para poder invertir un mapeo ninguna entrada puede tener valores nulos

// Para poder invertir un mapeo y que conserve su estructura debe cumplir la propiedad inyectiva de funciones, es decir no puede tener claves ni valores repetidos

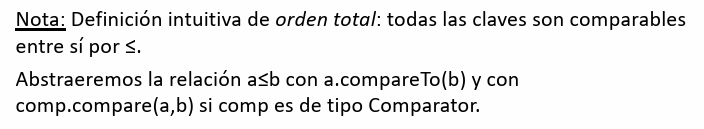
**TDA DICCIONARIOS:**

* **DEFINICION:** Es un conjunto de pares (k,v) donde:
* **K**  🡪 Es una clave
* **V** 🡪 Es el valor del elemento cuya clave es K

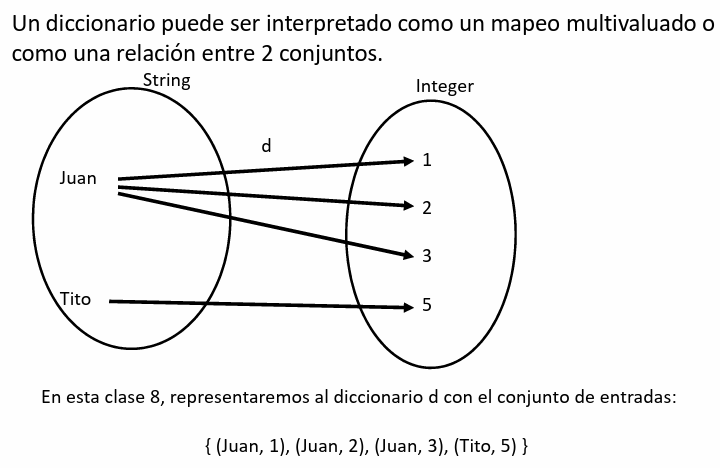
Cada uno de los pares se define como una entrada en el **diccionario e=(k,v)** [**K y V son elementos de tipo generico, Genericidad Parametrica**]

* **A diferencia del mapeo, pueden haber diferentes entradas con la misma clave** [coinciden algunas entradas con clave]
* **Al igual que los mapeos, un diccionario almacena pares de clave-valor**
* **Existen dos tipos de diccionarios: w**

1. **DICCIONARIOS NO ORDENADOS:** Las claves solo se comparan por igual [solo se comparan por equals]
2. **DICCIONARIOS ORDENADOS:** Hay una relacion de orden total definida para las claves [Se utiliza el comparador “”] [Para utilizacion con listas ordenadas; ordenados segun la clave de la entrada almacenada]



* **INTERPRETACION MATEMATICA DEL DICCIONARIO:**



VALOR [V]

CLAVE [K]

**DICCIONARIO:** Se utiliza este tipo de estructura cuando se quieren claves repetidas

**MAPEO GENERAL:** Se utiliza esta estructura cuando no se quiere entradas con claves repetidas

**MAPEO MULTIVALUADO:** Asocia una misma clave a distintos valores

* Un **TDADiccionario** se modela a travez de una interfaz que se encuentra especificada a traves de las siguientes operaciones **[Dado un diccionario ordenado D]**:

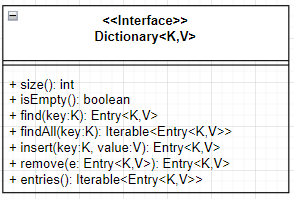
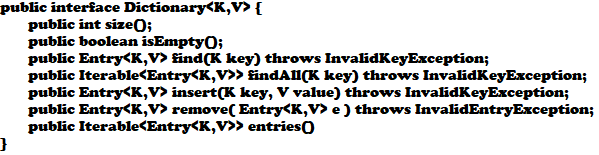
1. **Size()** 🡪 Retorna el número de entradas de D
2. **isEmpty()** 🡪 Testea si D está vacío
3. **Find(key)** 🡪 Si D contiene una entrada e con clave igual a key, entonces retorna e, sino retorna NULL **(\*)**
4. **FindAll(key)** 🡪 Retorna una colección iterable conteniendo todas las entradas con clave igual a key **[Retorna todos los valores asociados a dicha clave] [sino retorna coleccion iterable vacia]** **(\*)**
5. **Insert(key,value)** 🡪 Inserta en D una entrada e con clave key y valor value y retorna la entrada e **(\*)**
6. **Remove(e)** 🡪 Remueve de D la entrada e, retornando la entrada removida; ocurre un error si e no está en D **(\*)**
7. **Entries()** 🡪 Retorna una colección iterable con las entradas clave-valor de D **[Iterable de todas las entradas del diccionario; no se puede modificar el estado interno de un diccionario, solo puedo recorrerlo]**

**(\*) InvalidKeyException: Excepcion de clave invalida [clave nula]**

**(\*) NonExistentEntryException: Excepcion si la entrada no pertenece al diccionario**



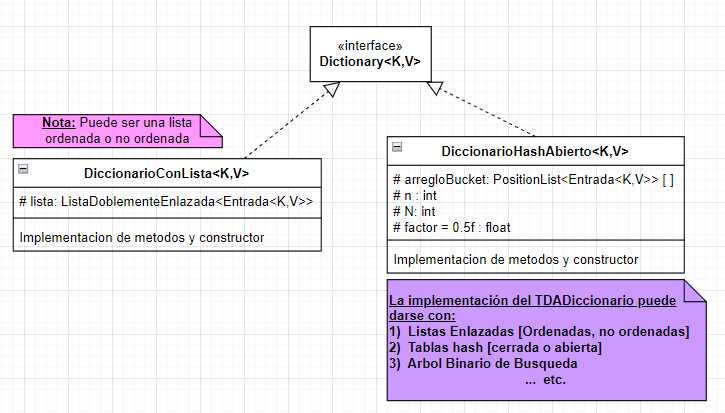
EJEMPLO DE LOS METODOS:



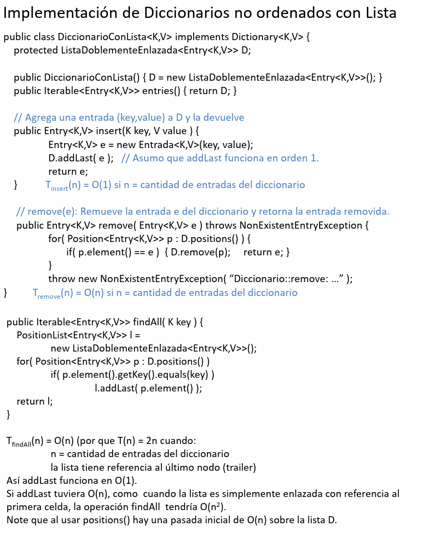
IMPLEMENTACION DE LA INTERFAZ DICCIONARIO EN JAVA:

DIAGRAMA UML DE LA INTERFAZ DICCIONARIO:

IMPLEMENTACIONES POSIBLES DE LA INTERFAZ MAP <K, V>:

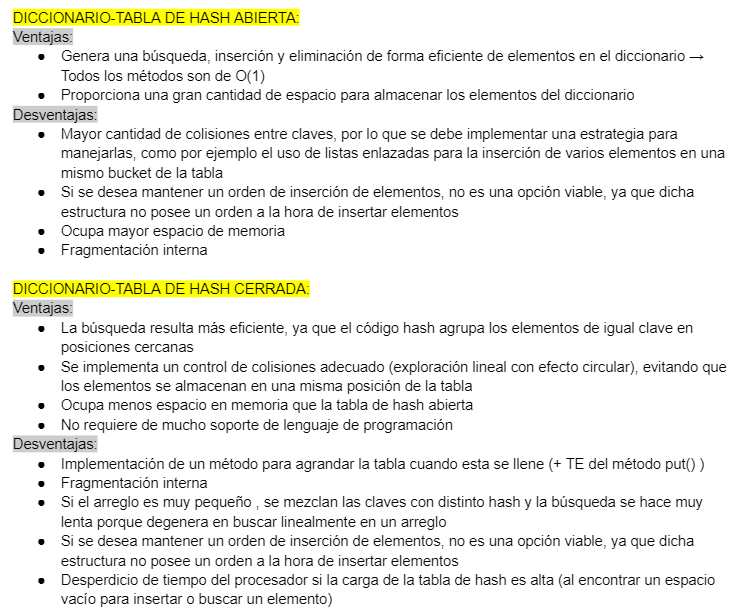


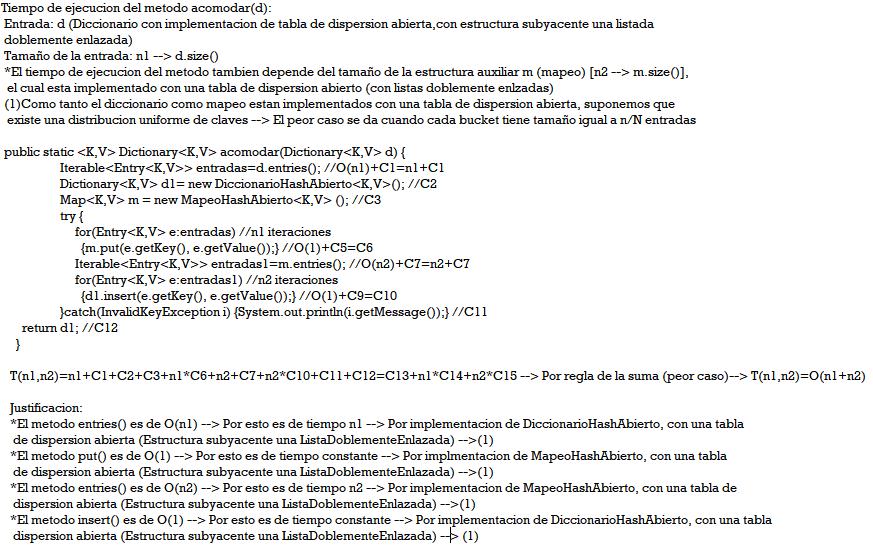
IMPLEMENTACION DE DICCIONARIO CON LISTA NO ORDENADA:



**//Ver implementacion de diccionario con tabla de hash Abierto [DiccionarioHashAbierto<K,V>]**

|  |  |  |
| --- | --- | --- |
| **METODOS** | **ORDEN DEL TIEMPO DE EJECUCION** | |
|  | **TIPO DE IMPLEMENTACION** | |
|  | **Con Lista no ordenada** | **Con hash Abierto** |
| **Size()** | **O(1)** | **O(1)** |
| **isEmpty()** | **O(1)** | **O(1)** |
| **Find(k)** | **O(n)** | **O(1)** |
| **findAll(k)** | **O(n2)** | **O(1)** |
| **Insert(k,v)** | **O(1)** | **O(1)** |
| **Remove(e)** | **O(n)** | **O(1)** |
| **Entries()** | **O(1) o O(n)** | **O(n)** |
| **reHash()** | **-** | **O(n)** |
| **hashThisKey(k)** | **-** | **O(1)** |

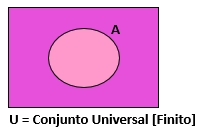




METODO QUE APLICA AMBAS ESTRUCTURAS [DICCIONARIO Y MAPEO]:

**TDA CONJUNTOS:**

* **DEFINICION:** Un conjunto se define como una estructura de datos que almacena una colección de elementos distintos. Es una estructura de datos **fundamental que se utiliza para almacenar y manipular un grupo de objetos, donde cada objeto es único**. El conjunto no permite elementos duplicados
* Un conjunto caracterizado con el tipo **Set<E> [TDA, Interfaz]** , tiene definida las operaciones:

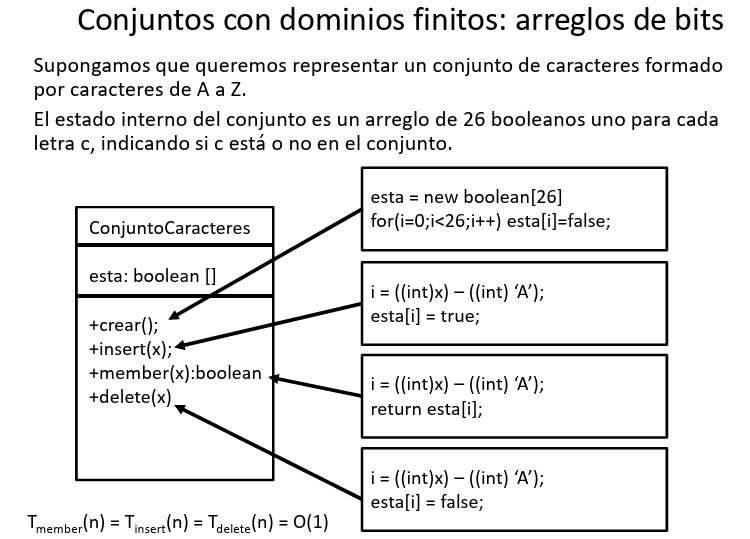
1. **Insert(x)** 🡪 Inserta un elemento x el conjunto
2. **Delete(x)** 🡪 Elimina el elemento x del conjunto
3. **Member(x)** 🡪 Retorna true si x pertenece al conjunto
4. **Intersection(S)** 🡪 Interseca al conjunto con otro conjunto S y retorna un nuevo conjunto intersección 🡪 por cada elemento x de this buscarlo en S y si está en S insertarlo en el resultado
5. **Union(S)** 🡪 Une al conjunto con el conjunto S y retorna un nuevo conjunto union



1. **Complement()** 🡪 Calcula y retorna el complemento del conjunto (solo se puede implementar si el dominio es finito) [Conjunto universal finito]
2. **Iterator()** 🡪 Retorna un iterador con los elementos del conjunto

**//Ver implementacion de la interfaz Set<E> [Lista enlazada o un mapeo de E en Boolean]**

|  |  |
| --- | --- |
| **METODO** | **ORDEN DEL TIEMPO DE EJECUCION** |
| **Insert(x)** | **O(1)** |
| **Delete(x)** | **O(n)** |
| **Member(x)** | **O(n)** |
| **Intersection(S)** | **O(n\*m) [Si this tiene n elementos y S tiene m elementos]** |
| **Union(S)** | **O(n+m\*n) [Si this tiene n elementos y S tiene m elementos]** |
| **Complement()** | **O(tamaño del dominio \* n)** |
| **Iterator()** | **O(1)** |





**EXPLICACIONES:**

1. Tengo un arreglo de 26 caracteres [alfabeto] 🡪 Inicializo en falso cada elemento del arreglo
2. Calculo índice del elemento x [código ASCII] 🡪 Asigno true a la componente del arreglo con índice (i) igual al código ASCII de x
3. Calculo índice del elemento x en el arreglo (código ASCII) 🡪 Retorno el valor boolean en la componente del arreglo con dicho índice
4. Calcula el índice del elemento x según su código ASCII 🡪 A la componente del arreglo con igual índice 🡪 Le asigno valor false

**Conjunto de arreglo finito (uso arreglo de bits)**

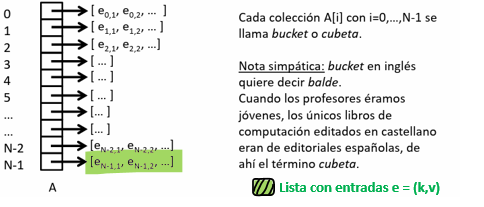
**TABLAS DE HASH:**

* **OBJETIVO DE LA IMPLEMENTACION:** Se desea tener una implementación versátil de conjuntos, diccionarios y mapeos donde las operaciones de inserción, recuperación y eliminación tengan un tiempo esperado constante
* **TIPOS DE IMPLEMENTACIONES POSIBLES:**

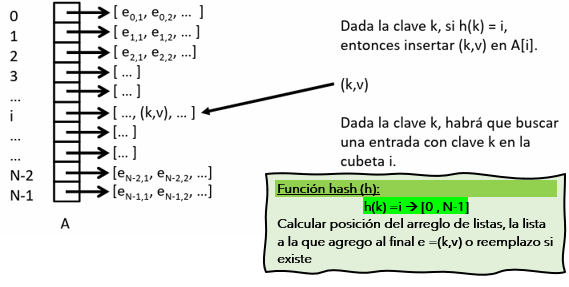
1. **TABLA DE HASH ABIERTO**
2. **TABLA DE HASH CERRADO**

**HASH ABIERTO:**

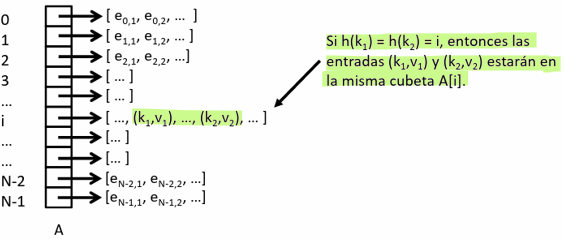
* Mantener por cada componente del arreglo una lista de entradas [Las colisiones determinan un elemento mas en cada lista]
* Un table de hash Abierto es un arreglo de buckets [cubetas]
* **ARREGLO DE BUCKETS:** Un arreglo de cubetas para implementar una tabla de hash es un arreglo A de N componentes, donde cada celda de A es una colección de entradas (pares clave-valor) eij



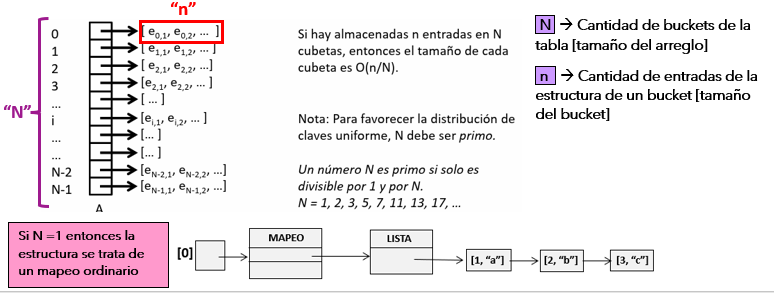
* **FUNCION DE HASH H:** Dada una clave k y un valor v, h(k) es un entero en el intervalo [0,N-1] tal que la entrada (k,v) se inserta en la cubeta A[h(k)]

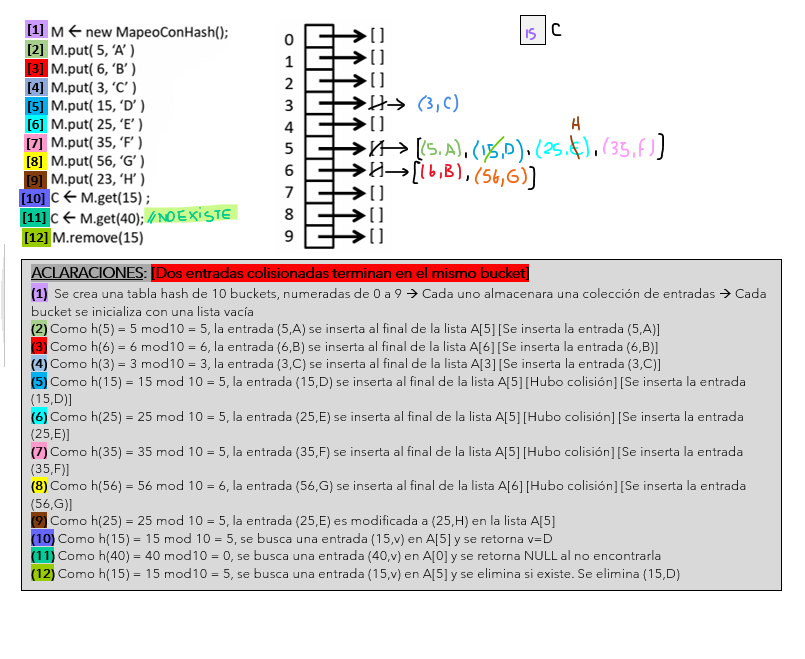


* **COLISION:** Dadas dos claves k1 y k2 tales que k1≠k2, se produce una colisión cuando h(k1) = h(k2) **[Las entradas de una cubeta tienen claves colisionadas] [Se minimizan utilizando la function de Hash]**

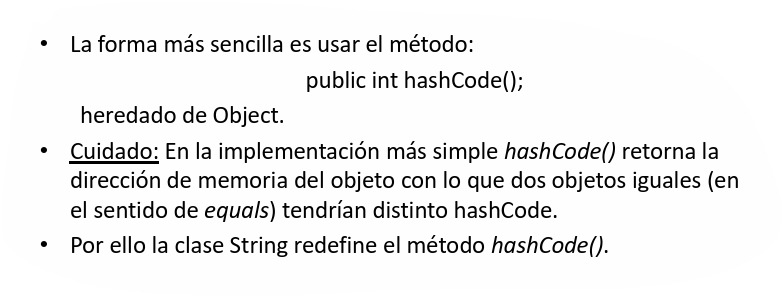


* **“BUENA” FUNCION DE HASH:** Una función de hash h es “buena” si h distribuye las claves uniformemente en el intervalo [0,N-1] 🡪 Intuitivamente, todas las cubetas tienen aproximadamente el mismo tamaño

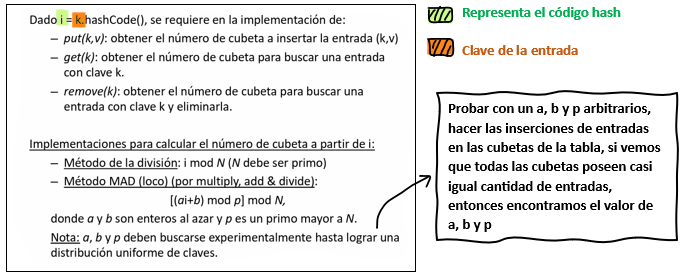
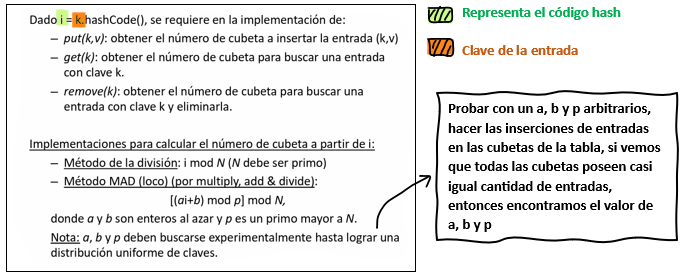


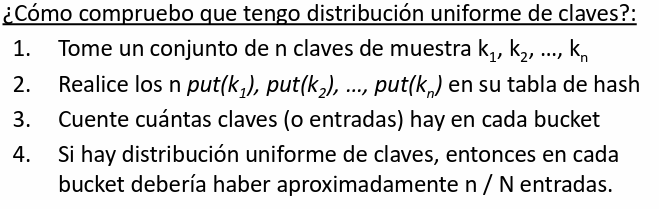
**EJEMPLO DE TABLA DE HASH ABIERTO [MAPEO]:**

* **HASHING CON CLAVES GENERICAS [Definicion de h(k) cuando k no es entero]:**
* **Paso 1 (hash code):** Dada una clave k, obtener un número entero llamado código de hash a partir de k



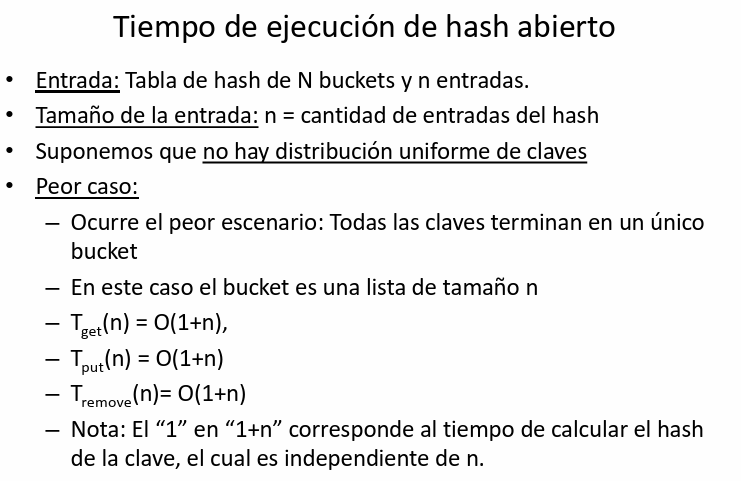
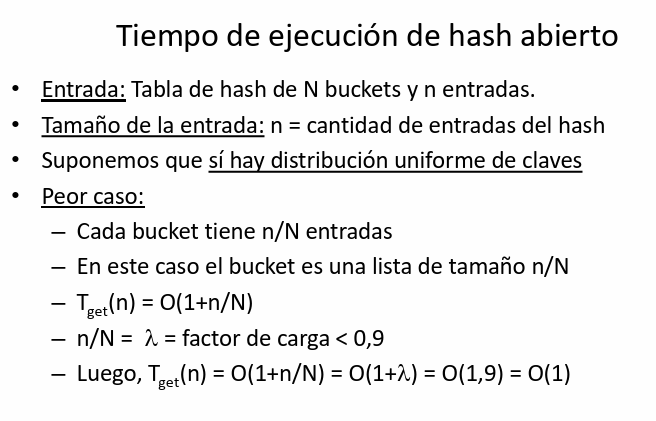
**//Genera código hash**

* **Paso 2 (función de compresión):** A partir del código de hash obtener un valor entero entre 0 y N-1 [Obtener numero de cubeta]
* **DISTRIBUCION UNIFORME DE CLAVES:**
* **Fenómeno aleatorio:** Fenómeno que bajo mismas condiciones puede producir resultados diferentes
* **Evento:** Resultado posible de un fenómeno aleatorio
* **Espacio muestral:** Conjunto de eventos posibles
* **Probabilidad:** La probabilidad de un evento E, notada p(E), es un número real entre 0 y 1
* **Frecuencia relativa:** La probabilidad de un evento E se estima con su frecuencia relativa = cantidad de ocurrencias del evento E / cantidad de eventos ocurridos
* **Distribución uniforme:** Todos los eventos tienen la misma probabilidad
* **Distribución uniforme de claves:** La probabilidad de que una clave termine en un bucket determinado es constante y es igual a 1/N, donde N=cantidad de buckets de la tabla de hash



**DISTRIBUCION: Distribución de la probabilidad de los eventos o la frecuencia en la que se dan los eventos**

TIEMPO DE EJCUCION DE HASH ABIERTO EN EL PEOR CASO:



TIEMPO DE EJCUCION DE HASH ABIERTO EN EL MEJOR CASO:

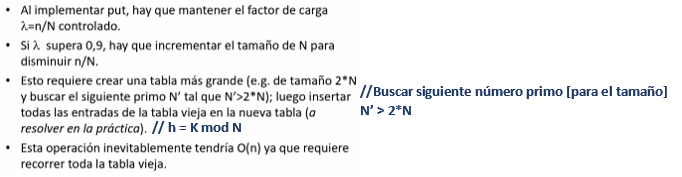
* El **FACTOR DE CARGA** cuando la distribucion de claves es uniforme, genera que todos los metodos de una implementacion sean de O (1)

**EJEMPLO DE APLICACIÓN DE FACTOR DE CARGA:**

Supongamos n= 200 y N = 8, entonces [Tope de cantidad de entradas por bucket, independiente de “n”; al generar que la cantidad de entradas por bucket sea constante, entonces los tiempos de recorrer, eliminar e insertar se dan en O(1)]

En este caso en el peor caso: get (), put() y remove() deben hacer 25 pasos, que es independiente de n y N, es decir es de O(1) [con una constante muy grande]

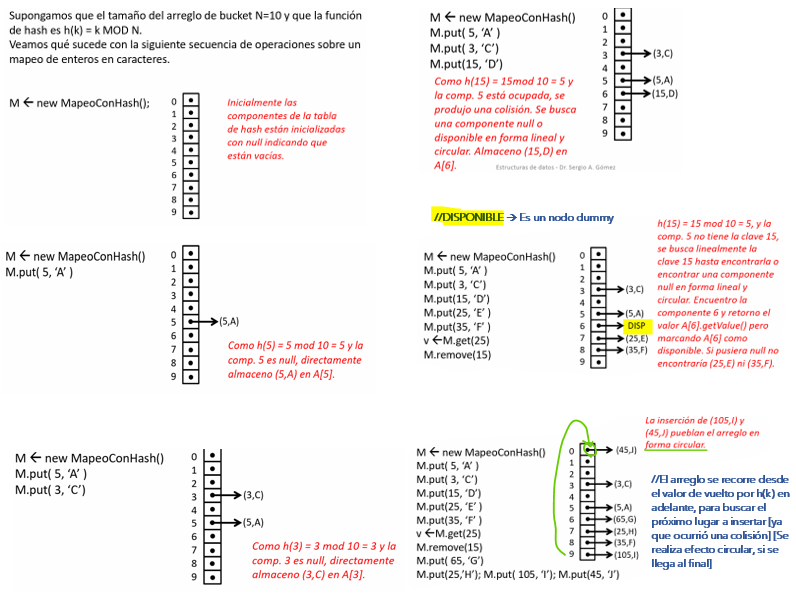
Si se inserta la entrada 201, para que la cantidad de entradas en un bucket no supere el límite establecido, aumento N para que 201/N < 25 =

* **REHASH:** Agrandar tabla de hash si n/N > factor de carga =0 ,9

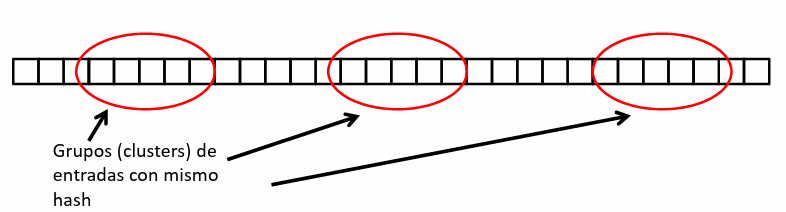
**HASH CERRADO:**

* **Almacenan directamente los elementos en cada contenedor** [Maneja un sistema de colisiones mas complejo]
* En hash cerrado se tiene un arreglo de N buckets, donde cada uno almacena a lo sumo una entrada
* Dada una clave k y un valor v, la **FUNCION HASH** h indica cuál es la componente h(k) del arreglo en la cual se almacena la entrada (k,v)
* Dadas dos claves k1 y k2 con k1≠k2, si h(k1)=h(k2) entonces se produce una **COLISION**
* **POLITICAS PARA LA RESOLUCION DE COLISIONES:**

1. **LINEAL [Mas facil de implementar , te garantiza siempre encontrar un lugar vacio]**
2. **CUADRATICA**
3. **HASH DOBLE**

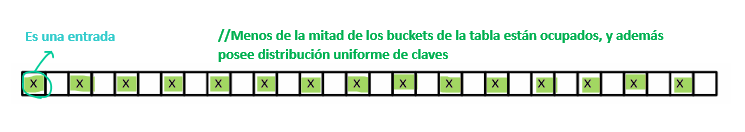
**EJEMPLO USANDO RESOLUCION LINEAL DE COLISIONES:**

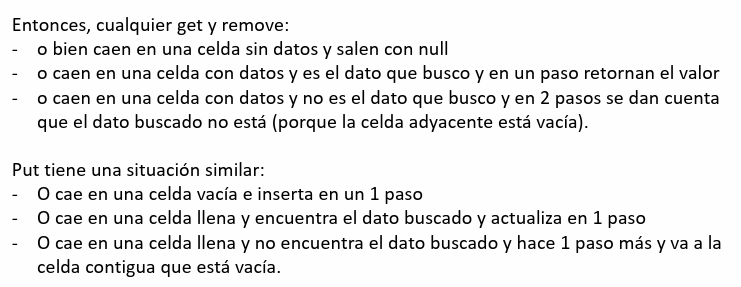
**VENTAJAS Y DESVENTAJAS DEL USO DE RESOLUCION LINEAL DE COLISIONES:**

* El hashing lineal tiende a agrupar entradas con claves con mismo valor de hash
* Si el arreglo es muy pequeño, se mezclan las claves ki con distinto hash y la busqueda se hace muy lenta porque degenera en buscar linealmente en un arreglo
* Para que esto funcione N >> n con lo que las claves se agrupan por su valor de hash [SUGERENCIA: factor de carga

**HASH CERRADO CON RESOLUCION LINEAL DE COLISIONES Y :**

* Si tengo distribución uniforme de claves y = 0.5, quiere decir que n/N<0.5 🡪 Es decir, que el peor escenario esperado es algo similar a esto (donde x representa datos cargados)



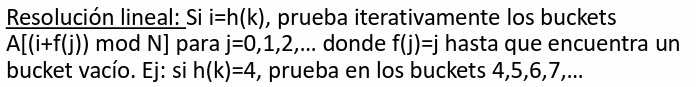


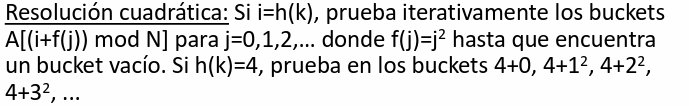


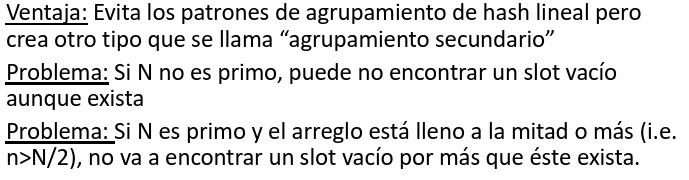
**[ O(1) ]**

**[ O(1) ]**

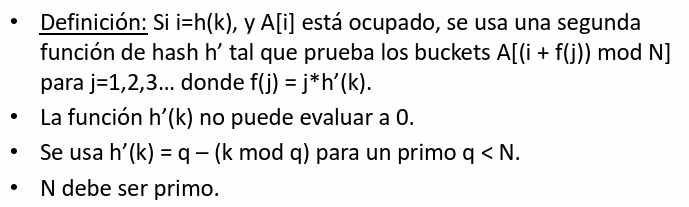




**OTRAS POLITICAS DE RESOLUCION DE COLISIONES: RESOLUCION CUADRATICA**



**OTRAS POLITICAS DE RESOLUCION DE COLISIONES: HASHING DOBLE**



COMPARACION ENTRE HASH CERRADO Y HASH ABIERTO:

* El hash cerrado ahorra algo de espacio con respecto al hash abierto.
* Los resultados experimentales muestran que el hash abierto es más rápido que los otros métodos
* El hash cerrado requiere muy poco soporte de lenguaje de programación