FACULTAD DE INGENIERIA

Universidad de Buenos Aires

TDA n3 - Tabla de hash

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	Vallejos, Matias
Número de padrón:	107924
Email:	matiavallejo@gmail.com
Email Institucional:	mvallejos@fi.uba.ar

1. Introducción

El objetivo de este TDA es implementar una solución simple y eficiente a la representación de estructuras de datos, en este caso se va a utilizar **Tablas de Hash** no solo por su estructura sino por su eficiencia a la hora de realizar operaciones tales como la búsqueda de datos y evitar tener elementos repetidos en la estructura. Este caso esta implementado mediante listas simplemente enlazadas esto nos va a ayudar a la hora de que tenga que colisionar un dato en el momento de insertarlo y debido a su facilidad a la hora de expandir o achicar estas listas.

2. Teoría

1. **¿Que es una tabla de hash?**

Una tabla de hash es una estructura que contiene **valores**, a estos valores yo los puedo acceder mediante una **clave**, para saber que posición debemos utilizar para insertar o hallar el valor se utiliza una **función hash** sobre la clave la cual las transforma en una posición en la tabla. Esta función hash puede generar colisiones (Cuando dos claves distintas tienen la misma posición), estas colisiones las resolveremos dependiendo el tipo de hash, en este proyecto utilizaremos una **tabla de hash abierta** con **direccionamiento cerrado**. Las tablas de hash son especialmente útiles a hora de realizar una búsqueda ya que estas cuentan con una complejidad O(n) en el peor de los casos y O(1) en el caso

promedio logrando así ser una de las estructuras más útiles cuando se almacenan grandes cantidades de información.

2. **Tipos de tabla de hash**

- **Hash abierto:** Son los que utilizaremos a lo largo del TDA. Estos hashes se dicen abiertos ya que la información se guarda "afuera" de la tabla de hash y poseen **direccionamiento cerrado** ya que la clave siempre va a estar en la posición que nos de la función hash. Este tipo de tabla resuelve las colisiones mediante encadenamiento, en este caso utilizando **listas simplemente enlazadas** por orden en que se insertan los valores. La complejidad de buscar en este hash suele ser O(n) ya que voy a tener que recorrer la lista enlazada.
- **Hash cerrado:** Estos hashes se dicen cerrado ya que todos los valores se guardan en la tabla y poseen **direccionamiento abierto** ya que el valor puede estar en una posición diferente a lo que me de la función hash con la clave. Debido a que estos tipos de hash no poseen listas y cada valor se guarda en la tabla entonces el tamaño de la tabla siempre va a tener que ser mayor o igual al número de claves. Este tipo de tabla para resolver las colisiones recorre el vector hasta encontrar el siguiente espacio libre. Hay diferentes tipos de método de búsqueda en este tipo de hash, algunos de ellos son:
  - Probing lineal: Busca el siguiente espacio libre inmediato.
  - Probing cuadrático: Toma el índice original del hash y añade sucesivamente valores de un polinomio cuadrático hasta que encuentra un espacio vacío.
  - Hash doble: Aplica otra función hash a la clave cuando hay colisión.

3. **Extra: ¿Por qué utilizar una lista simplemente enlazada?**

El uso de la lista es una solución sencilla al problema de las colisiones, dado que la función hash puede devolver índices similares para evitar que los elementos colisionen uso listas simplemente enlazadas y poder almacenar así elementos con el mismo índice de hash. Es gracias a esto que es un **Hash abierto** ya que los elementos no se guardan directamente en el hash, sino que en el hash se guardan listas que contienen a los elementos.

3. Detalles de implementación

Como compilar: Se puede utilizar el makefile incluido en los archivos con el comando make para tanto compilar las pruebas y ejecutarlas. **Archivos:** Cuenta con 4 archivos principales, hash.h el cual es la biblioteca general para utilizar el hash, hash.c el cual posee toda la implementacion realizada del hash, lista.h el cual posee la biblioteca general para utilizar las listas simplemente enlazadas hecha en TDAS anteriores y la cual utilizaremos para desarollar el hash, lista.c la cual posee toda la implementacion de las listas simplemente enlazadas. Como adicional tambien se encuentra un archivo pruebas.c las cuales contiene todas las pruebas necesarias para probar el correcto funcionamiento del hash

1. `destruir_tabla`

   Es utilizada para destruir el hash en caso de error o en el caso de tener que rehashear la tabla.

2. `funcion_hash`

   Es la función hash, devuelve el índice que se utilizara para insertar/buscar/quitar elementos. Para calcularlo lo que hace es ir sumando las letras del string clave (Esto lo hace ya que cada letra tiene una representación numérica en ASCII) y devuelve el resto de dividir esa suma con la capacidad.

3. `rehash`

   Va a aumentar la capacidad de la tabla, en este caso lo duplica. Lo hace mediante la creación de una tabla auxiliar con capacidad mayor, copia los datos de la tabla anterior a esta nueva auxiliar y finalmente elimina la tabla anterior. Utiliza las funciones de lista iterador del TDA lista para iterar sobre la lista y así insertarlas en el nuevo hash y luego eliminarlas del antiguo. (Ver diagrama "Rehash").

4. `buscar_elemento`

   Busca un elemento en el hash, un elemento es el struct dentro de las listas del hash, los elementos contienen la clave y el valor. Utiliza la función hash para posicionarse en la lista y luego la función iterador para recorrer esa lista. Devuelve el elemento encontrado o NULL en caso de que falle o no lo encuentre. Esta función se utiliza luego para sobrescribir un valor con clave repetida en el hash, para quitar un elemento del hash, para obtener un elemento (así se puede devolver el valor) y para ver si el hash contiene a ese elemento.

5. `copiar_clave`

   Devuelve una copia de la clave reservada en el heap con malloc. Sirve para no tener que usar una clave por referencia.

6. `modificar_valor_hash`

   Utiliza la función buscar_elemento para buscar el elemento con la misma clave en el hash y modificarle su valor. Devuelve 0 si tuvo exito 0 -1 si fallo.

7. `hash_insertar`

   Inserta un elemento en el hash. Primero chequea si es necesario rehashear, si no es necesario va a aplicar la función hash para buscar una lista en donde insertarlo antes de insertarlo chequea si esta clave ya está en el hash ya que, si está en el hash solo se debería modificar su valor y no volverla a insertar (Esto porque **no hay duplicados en el hash**) en el caso en el que no esté en el hash reserva memoria en el heap para un elemento_t* le asigna su valor y **una copia de la clave** esto se debe a que si no se asignaría la clave por referencia y generaría conflictos a la hora de liberarlos. Finalmente inserta ese elemento en la lista. Devuelve 0 en caso de éxito o -1 en caso de error. Si inserta un elemento aumenta su cantidad. (Ver diagramas "Insertar", "Insertar con colisión", "Insertar con clave repetida", "Insertar con rehash")

> Rehash: Se calcula cuando es necesario rehashear mediante dividir la cantidad de elementos en el hash sobre la capacidad de elementos en el hash, si esta es mayor o igual a 0.75 entonces se rehashea.

1. `buscar_posicion_en_lista`

   Recibe un hash y una clave, busca primero la posición de esa clave en el hash luego con esa posición busca en la lista la posición en donde está la clave en la lista y la devuelve. Utiliza la función hash para encontrar la lista y las funciones de iteración de la lista para encontrar la posición. Devuelve 0 en caso de no encontrarla o error.

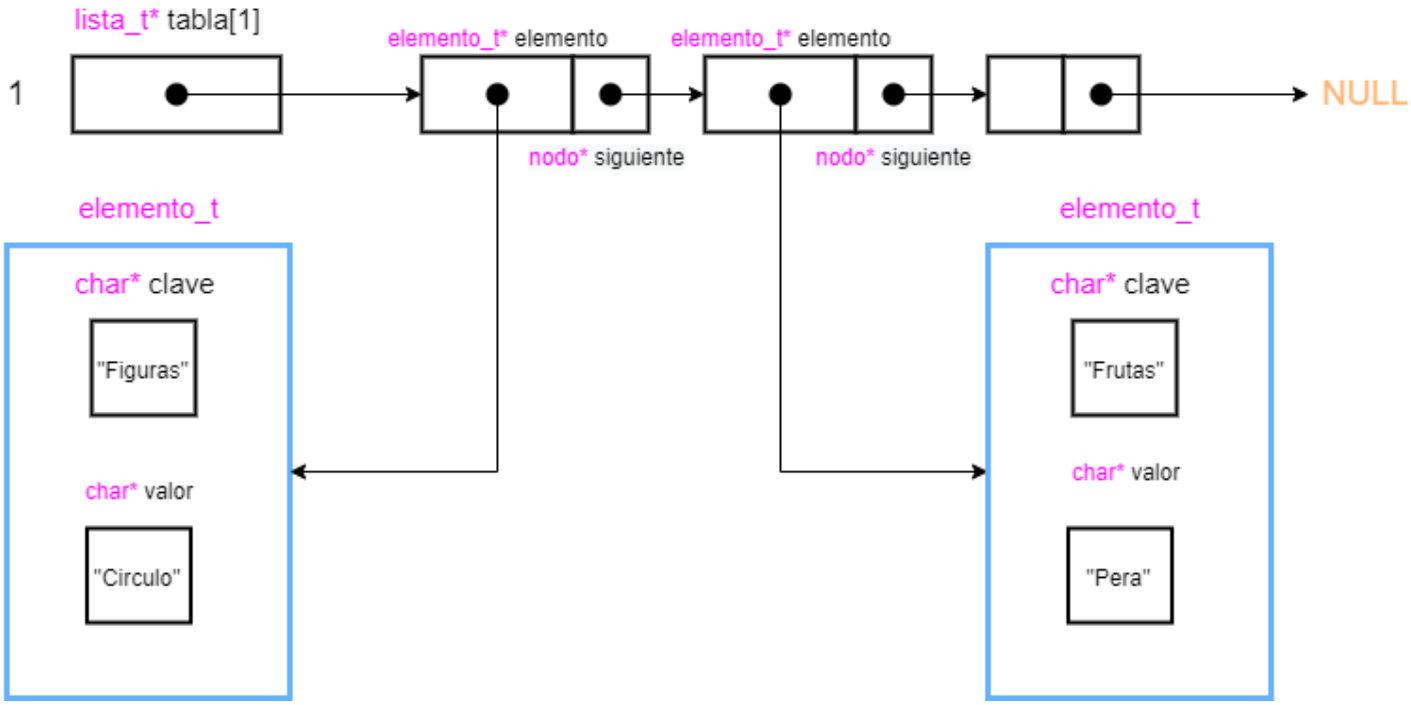
2. `hash_quitar`

   Recibe un hash y una clave, utiliza la funcion_buscar elemento para buscar el elemento que tiene que remover, la función hash para encontrar la lista en la que lo tiene que remover, la función buscar_posicion_en_lista para encontrar la posición de la lista en la que está el elemento a remover. Utiliza finalmente la función lista_quitar_de_posicion para remover el elemento de la lista y finalmente aplica el destructor si es que hay y libera la clave y el elemento, disminuye la cantidad de elementos. Devuelve 0 en caso de éxito o -1 en caso de error. (Ver diagrama "Quitar")

4. Diagramas

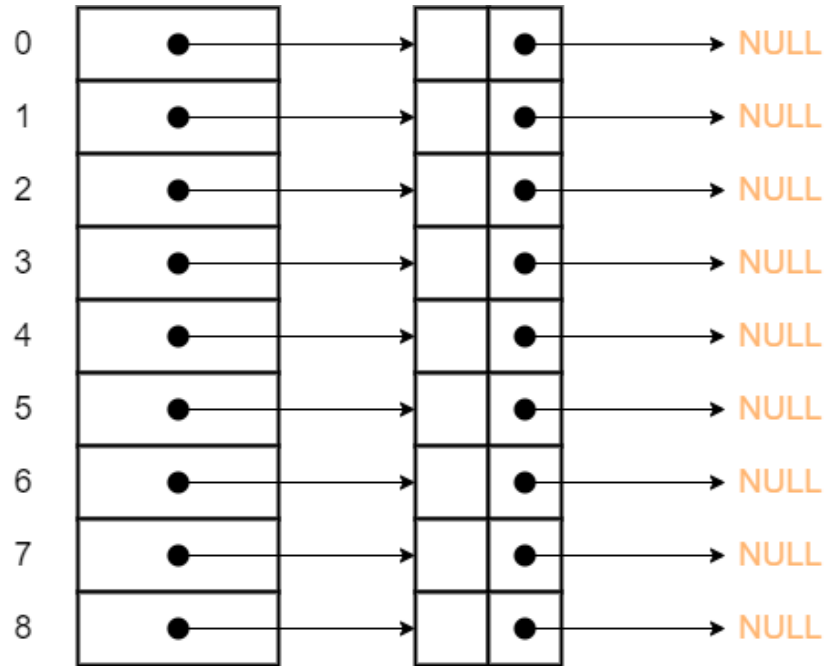
Nota sobre los diagramas: No son 100% representativos de como esta desarrollado el hash en el código, sino que solo guardo el valor en las listas, ya que si fueran así en vez de guardar solo el valor en el hash tendría que guardar un puntero elemento_t el cual es un struct que a su vez es un puntero hacia la clave y el valor, y serian diagramas muy grandes, este es un ejemplo un poco más realista:

Clave	Hash	Valor
"Figuras"	1	"Circulo"
"Frutas"	1	"Pera"



1. Creación del hash

Clave	Hash	Valor

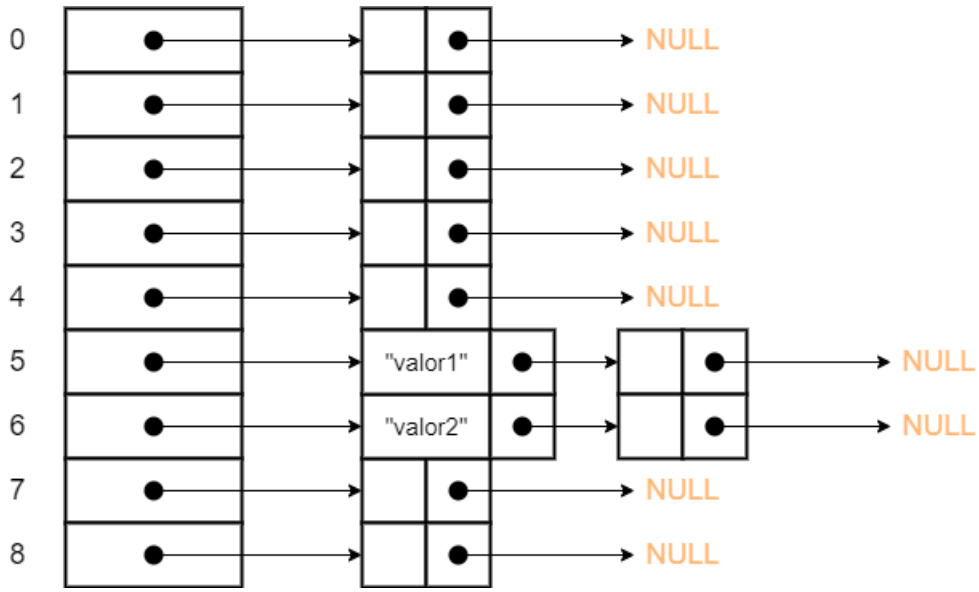


Este diagrama muestra la creación de un hash vacío, puede notarse que cada posición del vector apunta hacia una lista simplemente enlazada donde no contiene ningún elemento y su siguiente es NULL. En la tabla tenemos la clave y el valor el cual es definido por el usuario y el hash es la

posición que nos devuelve la función hash sobre la clave, esta también va a ser la posición en donde buscaremos la lista simplemente enlazada para ingresar nuestro valor. En este caso la tabla tiene una capacidad inicial de 9 elementos.

2. Insertar

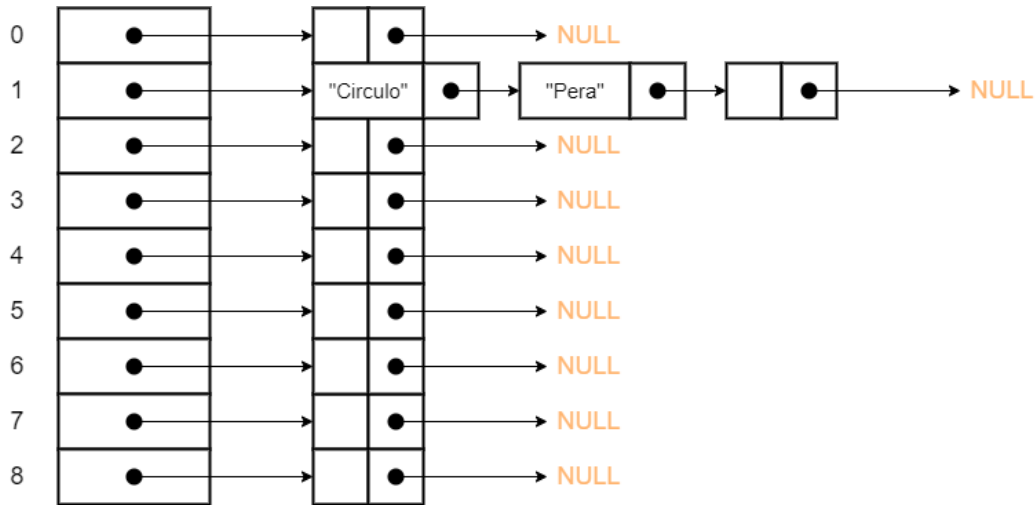
Clave	Hash	Valor
"clave1"	5	"valor1"
"clave2"	6	"valor2"



Para insertar un elemento utiliza la función hash la cual mediante el uso de la clave y la capacidad del hash obtiene un valor para el hash, en este caso si aplico la función a clave1 la función hash nos devuelve 5 y en clave2 nos devuelve 6, todo esto con un hash con capacidad 9. Para insertarlos en el hash simplemente voy a la posición 5, luego al primer elemento de la lista y como está vacío lo inserto ahí, lo mismo con la clave2.

3. Insertar con colisión

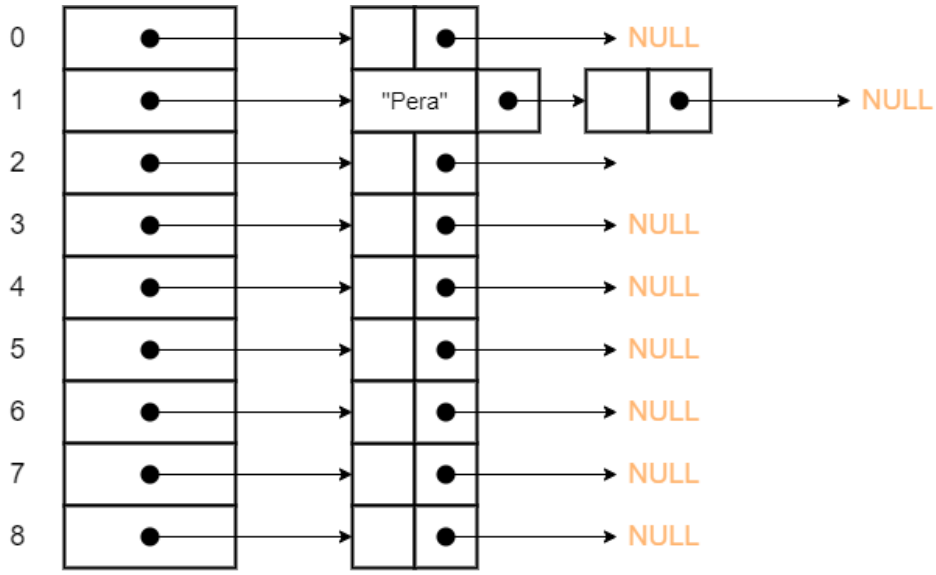
Clave	Hash	Valor
"Figuras"	1	"Circulo"
"Frutas"	1	"Pera"



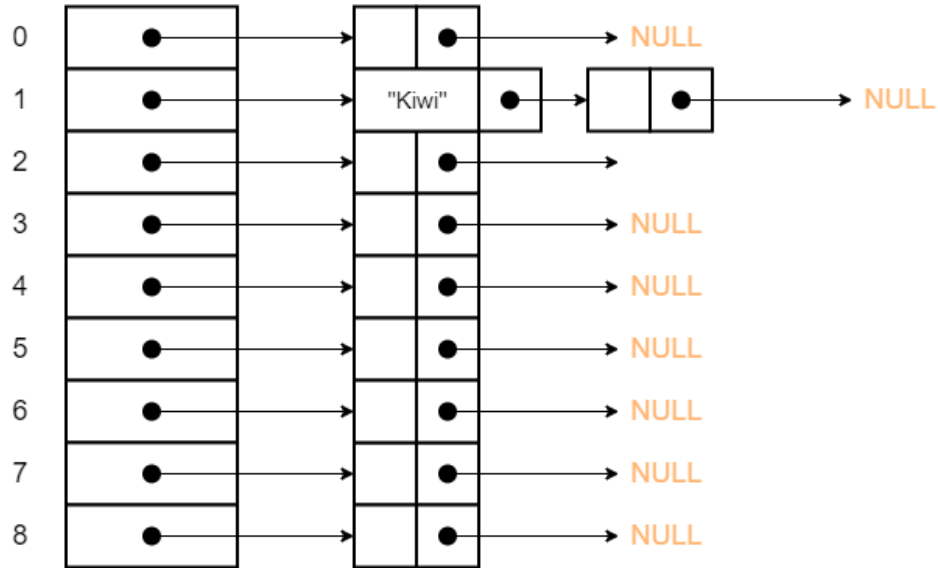
En este caso podemos observar que es lo que sucede cuando tenemos claves diferentes pero que luego de aplicar la función hash nos devuelve el mismo valor para la tabla, dado que en este TDA utilizamos un **Hash abierto** no tenemos mucha complicación a la hora de las colisiones ya que solamente se agrega un elemento extra en la lista. Hay que recordar que esto solo sucede si tenemos diferente clave.

4. Insertar con clave repetida

Clave	Hash	Valor
"Frutas"	1	"Pera"
"Frutas"	1	"Kiwi"



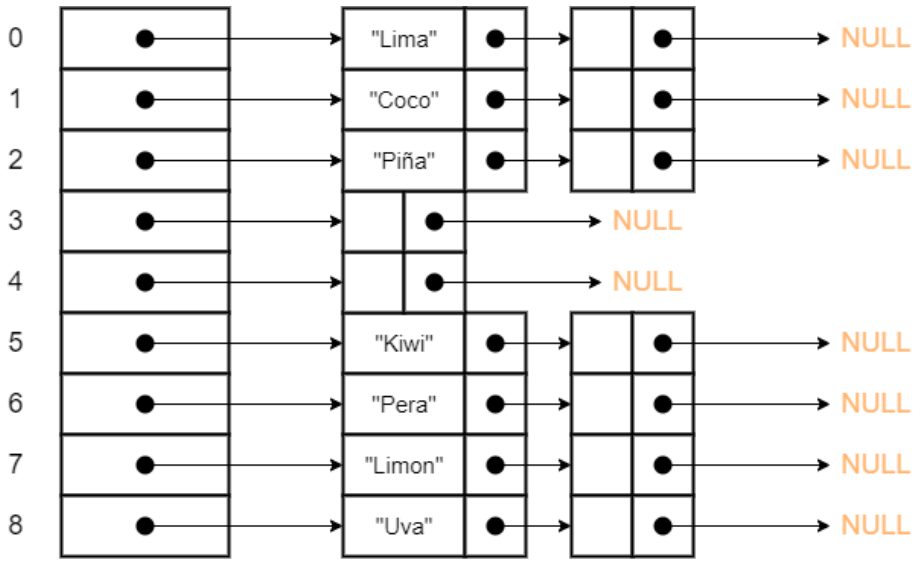
Clave	Hash	Valor
"Frutas"	1	"Pera"
"Frutas"	1	"Kiwi"



En este caso tenemos dos claves iguales, pero con diferente valor, dado que una de las características del hash es que las claves son únicas esto significa que **no podemos tener claves repetidas**. Dado que las claves son iguales la función hash va a devolver lo mismo, entonces primero se va a insertar la clave "Frutas" con valor "Pera" en la lista de la posición 1 y luego al intentar insertar la clave "Frutas" con el valor "Kiwi" dado que tienen la misma clave solo se va a sobrescribir el valor de "Pera" a "Kiwi", siendo "Kiwi" el valor final que va a quedar en la tabla.

5. Insertar con rehash

Clave	Hash	Valor
"Fruta1"	5	"Kiwi"
"Fruta2"	6	"Pera"
"Fruta3"	7	"Limon"
"Fruta4"	8	"Uva"
"Fruta5"	0	"Lima"
"Fruta6"	1	"Coco"
"Fruta7"	2	"Piña"
"Fruta8"	3	"Melon"
"Fruta9"	4	"Fresa"



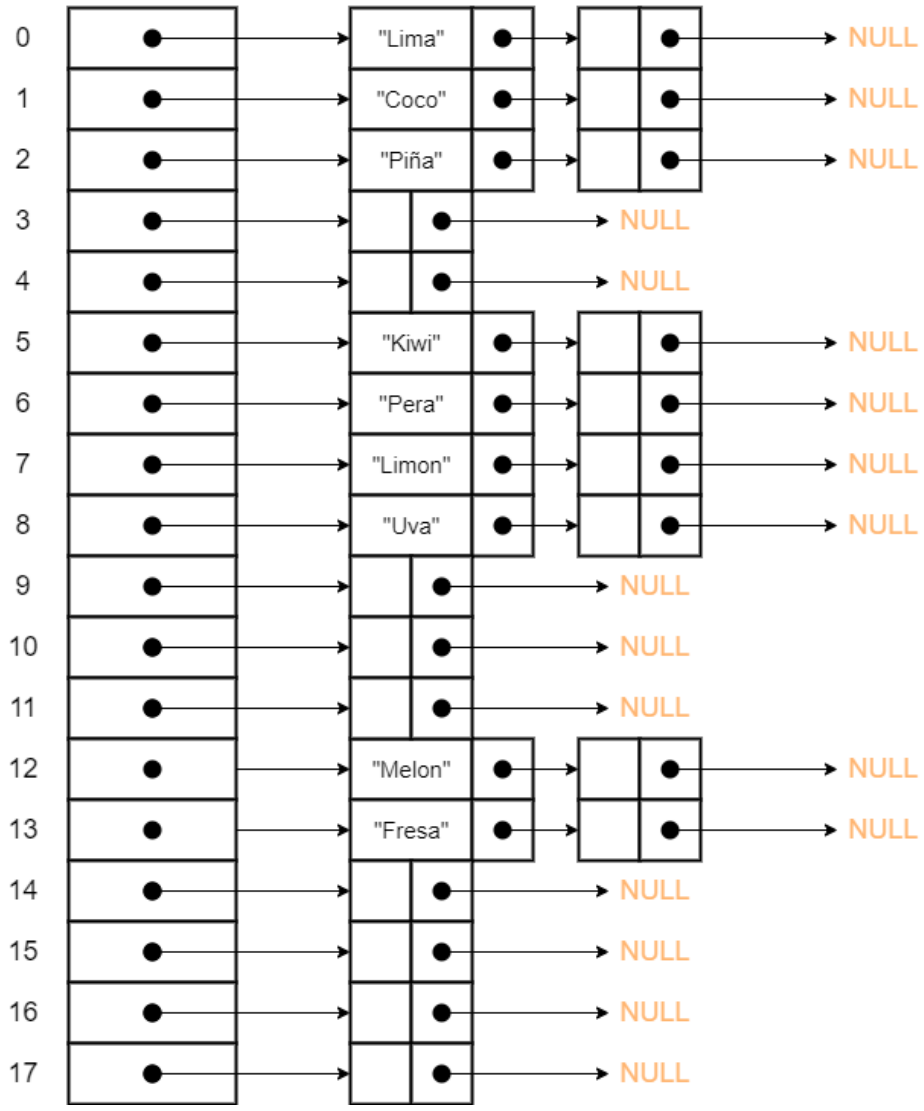
Factor de carga = Cantidad/Capacidad → 9

→ 7

Factor de carga predeterminado = 0.75

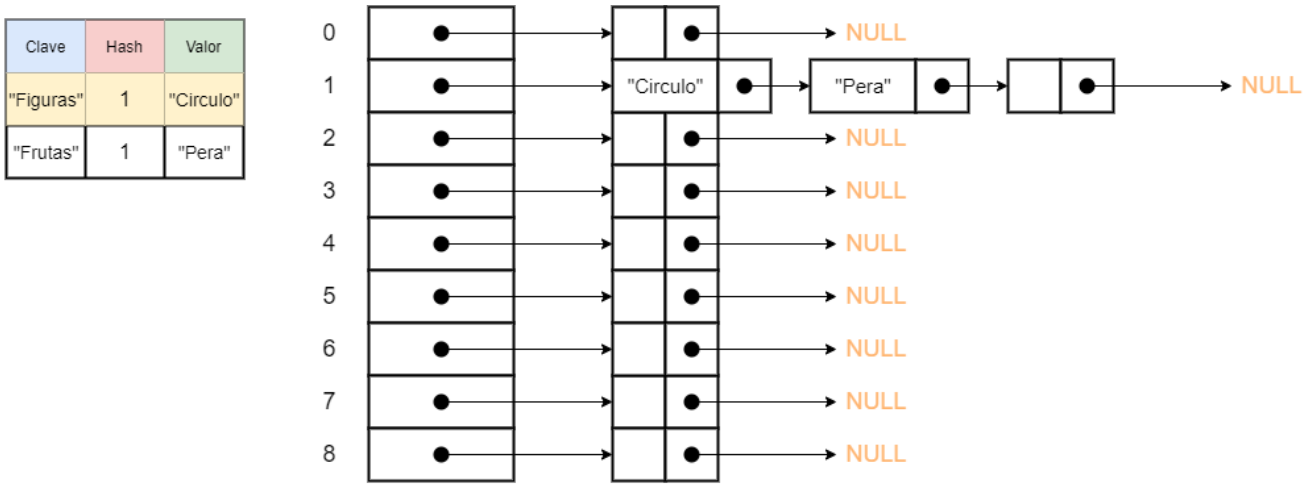
Clave	Hash	Valor
"Fruta1"	5	"Kiwi"
"Fruta2"	6	"Pera"
"Fruta3"	7	"Limon"
"Fruta4"	8	"Uva"
"Fruta5"	0	"Lima"
"Fruta6"	1	"Coco"
"Fruta7"	2	"Piña"
"Fruta8"	12	"Melon"
"Fruta9"	13	"Fresa"

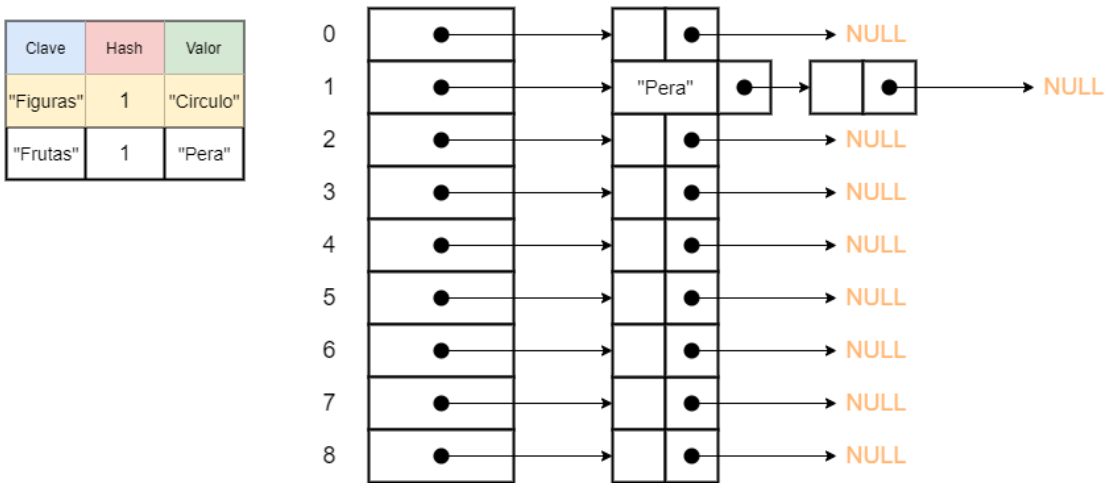
Capacidad → 18



En este caso vemos que pasa en caso de rehash, en la implementación del TDA se utiliza el factor de carga predeterminado(0.75), para calcular si es necesario rehashear tenemos que hacer la cantidad de elementos insertados/capacidad del hash, en este diagrama vemos que cuando el algoritmo inserta la clave "Fruta7" la cantidad va a seguir siendo 6 así que por lo tanto la va a insertar sin problemas sin embargo al llegar a la clave "Fruta8" la cantidad va a ser 7, como la capacidad es 9 el factor de carga nos da 0,78 el cual es mayor al factor predeterminado, eso significa que hay que hacer rehash. El rehash consiste en "agrandar" la tabla en este caso mediante la duplicación de su capacidad, o sea que nos va a quedar una tabla con capacidad final de 18 elementos, una vez rehasheamos insertamos los elementos faltantes normalmente hasta que otra vez tengamos que volver a rehashear. El rehash sirve para **Aumentar el rendimiento del hash** dado que este rendimiento va disminuyendo a medida que insertamos elementos, sin embargo, hay que tener en cuenta que rehashear es una operación **Muy costosa** es por eso por lo que no podemos recurrir al rehash en todo momento y elegimos un factor de carga ideal.

6. Quitar





El caso de quitar es sencillo ya que utilizamos la función `lista_quitar_de_posicion` implementada en el TDA de listas enlazadas.