TDA - HASH

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	**SANCHEZ FERNANDEZ DE LA VEGA, Manuel**
Número de padrón:	107951
Email:	msanchezf@fi.uba.ar

1. Introducción

El TDA Hash consiste principalmente de una tabla. Esta contiene pares de clave y valor. El valor es el elemento que deseamos guardar, y la clave es una cadena de caracteres que se traduce a una posición en la tabla mediante la función de hash.

Una de las ventajas del hash es que permite obtener elementos del mismo con un orden de complejidad de `O(1)`, al menos si esta implementado correctamente (mas detalle en la sección de teoría).

Hay diversas formas de implementar un hash, en las secciones posteriores explicare mi implementación y las decisiones detrás de la misma.

2. Teoría

Como ya lo mencione previamente, una tabla de hash es, esencialmente, una tabla de datos que consisten de pares clave y valor. Estas claves se traducen en posiciones en la tabla de hash.

Por ejemplo, yo puedo guardar un elemento con la clave `"abc"`, la cual mediante la función hash, es traducida a un numero que representa la posición en esta tabla. La ventaja de esto es que si yo quiero recuperar el elemento que guarde con la clave `"abc"`, es tan simple como volver a traducir la clave, y acceder a esa posición de la tabla. Como no es necesario iterar sobre toda la tabla, lo podemos considerar como `O(1)`.

Una consecuencia de el uso de estos pares clave y valor es que no pueden existir 2 valores con la misma clave, ya que las claves deben ser únicas. De otro modo, no habría forma de diferenciar cual estamos buscando.

A la hora de insertar elementos, puede suceder que 2 claves distintas se traduzcan en la misma posición en la tabla. A esto se lo conoce como colisión. Este problema es lo que genera una distinción entre los tipos de hash. Como consecuencia del mismo, existen 2 tipos:

- Hash abierto - direccionamiento cerrado
- Hash cerrado - direccionamiento abierto

Hash Abierto - direccionamiento cerrado

Los hash abiertos consisten en que los elementos no son insertados en la tabla de hash directamente, por eso se los llama abiertos. Técnicamente, los elementos están fuera de la tabla de hash. Por el otro lado, se lo conoce como direccionamiento cerrado ya que los elementos son insertados en la posición que devuelve la clave hasheada.

Los hash abiertos consisten en implementar algún otro TDA en la tabla de hash, como puede ser una lista o un árbol binario. Es decir, la tabla de hash consiste de un vector que en cada posición contiene algún TDA. Lo que nos permite hacer esto es que, si tenemos una colisión, los elementos pueden compartir esa posición de la tabla ya que se pueden insertar en el TDA.

Consecuentemente, para acceder a un elemento en un hash abierto, hay que acceder a la posición de la clave hasheada, y luego iterar en el TDA hasta encontrar el elemento con la clave buscada.

No obstante, esto puede traer problemas ya que si se insertan demasiados elementos en una sola posición de la tabla, el orden de complejidad de esta iteración puede dejar de ser considerado como insignificante por lo que perdemos eficiencia. Como solución, se suele rehashear. Es decir, agrandar el hash e insertar todos los elementos nuevamente (los cuales puede que caigan en posiciones distintas a las que estaban previamente). Sin embargo, se evita hacer esta operación seguido ya que es muy costosa.

Para eliminar elementos, simplemente se lo quita del TDA usando la primitiva correspondiente y listo.

Esta es la implementación que yo utilice y voy a detallar en la sección de mas abajo.

Hash Cerrado - direccionamiento cerrado

Por el otro lado, los hash cerrados se destacan por el hecho de que los elementos siempre son insertados en la tabla de hash, sin hacer uso de otro TDA de por medio. Los pares clave y valor siempre son insertados en la tabla, y si en algún momento ocurre una colisión, hay diversas formas de tratarla como:

- Probing lineal
- Probing cuadratico
- Hash doble

El probing lineal, por ejemplo, consiste en avanzar una posición en la tabla e intentar insertarlo ahí. Este proceso se repite hasta que se encuentra una posición libre. De ahí viene el nombre de direccionamiento abierto, ya que la posición del elemento no es siempre la que devuelve la función de hash.

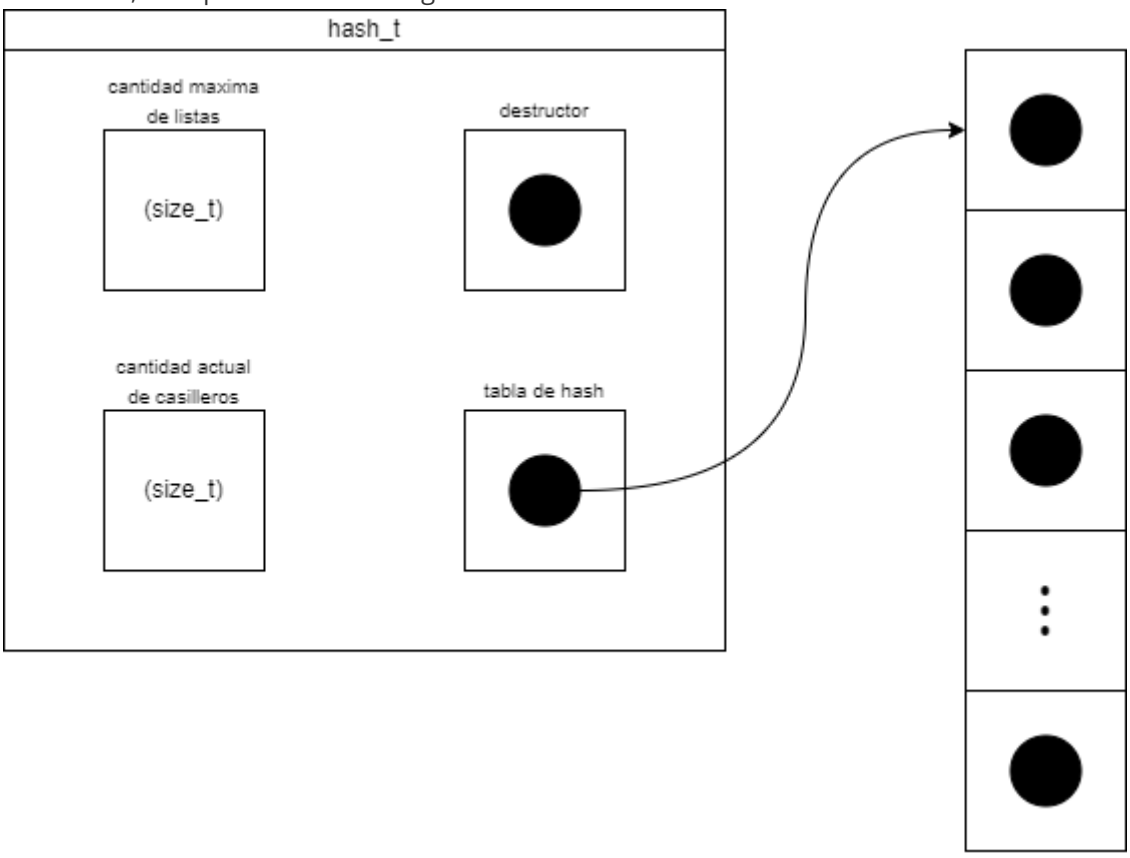
Si se intenta acceder a un elemento que no esta en su posición correspondiente, lo que se hace es avanzar de posición y comparar la clave. Este proceso se repite hasta que no haya mas claves por comparar, o hasta encontrarse con un espacio en blanco. Esto trae como consecuencia que, a la hora de eliminar un elemento, no es tan sencillo como quitarlo de la tabla ya que afectaría

otras búsquedas. Para solucionar esto, se pueden tomar enfoques distintos, pero uno de los mas simples es marcar al casillero con un flag que indique que tuvo un elemento eliminado. De esta forma, se evita afectar futuras búsquedas. Nuevamente, en esta implementación es necesario rehashear para evitar quedarse sin espacio para nuevos elementos.

Otra implementación posible es usar una zona de desbordes, la cual es una parte de la tabla de hash reservada para elementos colisionados. Si llegara a ocurrir una colisión, el elemento se agrega en la zona de desbordes en vez de realizar probing lineal. Esto produce que, a la hora de buscar un elemento, también debo buscar sobre la zona de desbordes.

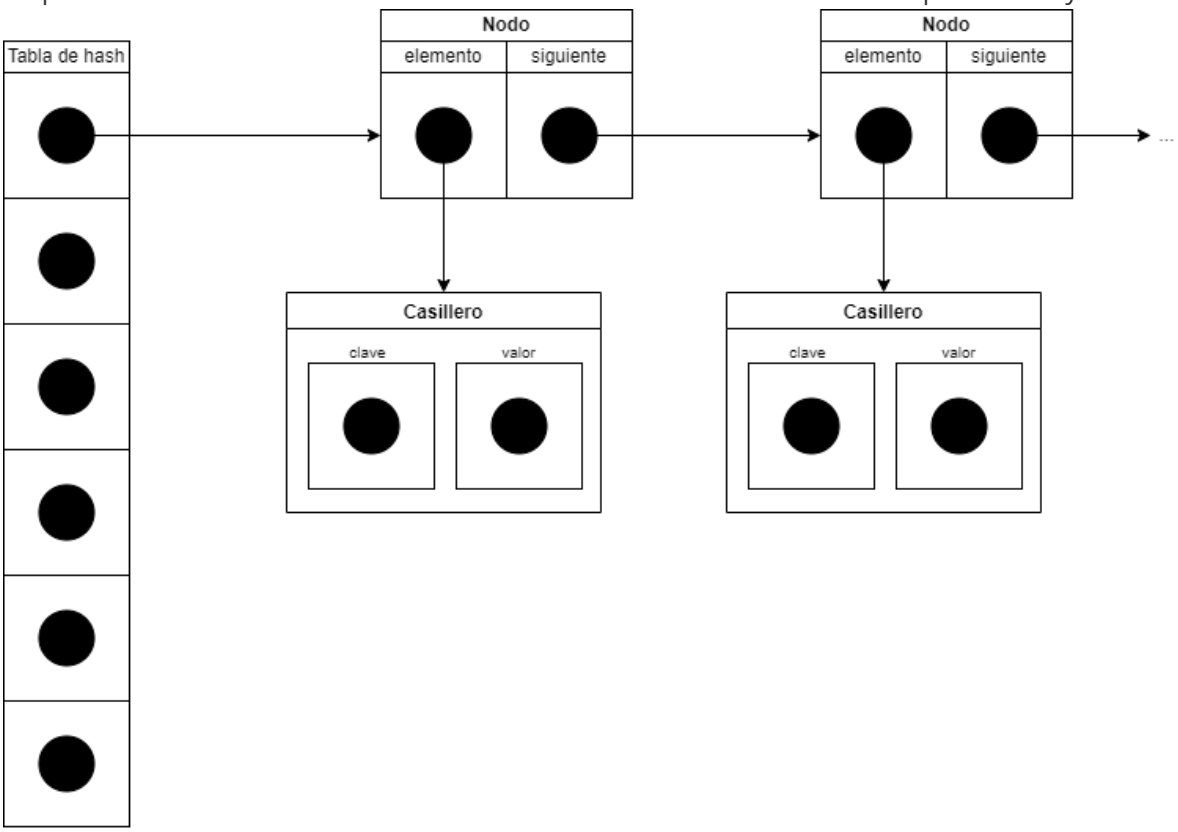
3. Detalles de implementación

En mi implementación, desarrolle un hash abierto. Es decir, mi tabla de hash consiste de un vector dinámico de listas simplemente enlazadas. En estas listas es donde se almacenan los pares clave y valor del hash. Opte por usar una lista antes que un ABB ya que me pareció una solución mas genérica y simple de implementar. Asimismo, los ABB necesitan una función de comparación para insertar elementos. Además, la ventaja de un ABB es la eficiencia que tiene frente a los tiempos de búsqueda de una lista, sin embargo, considerando que estos no van a tener muchos elementos, el impacto no será tan grande.



Decidí almacenar la cantidad actual de casilleros para saber cuantos elementos tiene el hash en todo momento. Por el otro lado, también guarde la cantidad máxima de listas. Estos parámetros son los que voy a usar para decidir cuando rehashear.

Yo decidí agrupar los pares clave y valor en una estructura que llame casillero. En otras palabras, lo que realmente contienen las listas son casilleros donde se almacenan los pares clave y valor.



En los casilleros, las claves son strings, es decir `char*`, mientras que los valores son punteros genericos.

De esta forma, las colisiones no son un problema ya que si 2 claves pertenencen a la misma posicion en la tabla, simplemente las inserto en la lista de esa posicion sin tener problema alguno.

Al estar trabajando con listas, puedo hacer uso de sus primitivas por ejemplo a la hora de insertar, quitar elementos, iterar sobre el hash, etc.

Voy a detallar el funcionamiento y decisiones implementación para algunas funciones que considero importantes:

1. `funcion_hash()`
   Hay infinidades de funciones de hash, algunas de ellas muy complejas pero eficientes. Sin embargo, para mi implementación opte por realizar una función muy sencilla que simplemente suma el valor de la tabla ASCII de cada carácter en la clave, y lo devuelve. Aunque es cierto que no es la función mas efectiva de todas (no realice ningún tipo de estudio de dispersión sobre la tabla ni nada), cumple su rol y si en algún momento decido cambiarla, es tan simple como modificar la `funcion_hash()` y listo. En otras palabras, el resto del código no se ve afectado.

2. `hash_insertar()`
   La parte que me gustaría destacar de esta función es el rehasheo. Si se intenta insertar un elemento y la cantidad de elementos del hash supera la cantidad de listas del hash multiplicado por una constante (MULTIPLICADOR_LIMITE_REHASHEO), entonces se realiza un rehash. Decidí definir a esta constante que multiplica la cantidad de listas como 5. Esto se debe a que si no pusiera un limite para el rehasheo, entonces las operaciones de búsqueda en el hash dejarían de ser O(1). Entonces, decidí que la capacidad máxima de casilleros en el hash es 5 veces la cantidad de listas que el mismo tiene.

3. `rehashear()`
   Esta función se resume en crear un hash nuevo con una cantidad de listas `multiplicador_tamanio` veces mas que el anterior hash. Luego copia todos los elementos del viejo hash al nuevo. Esta cantidad de veces mayor esta definida en una constante llamado MULTIPLICADOR_NUEVO_TAMANIO que definí en 10. Es decir, cada vez que rehasheo, el hash se hace 10 veces mas grande. Esto es así ya que, al ser una operación muy costosa, prefiero utilizar mas memoria a cambio de tener que hacer esta operación repetidas veces.

   Para desarrollar esta función, use el iterador interno del hash, pasándole como parámetro una función que copie los casilleros de uno a otro. Esta función se llama `copiar_casillero()` y devuelve false si lo pudo copiar y true si no pudo. Esto se debe a que el iterador interno continua iterando mientras que la función devuelva false. Asimismo, esta función es necesaria ya que no puedo usar directamente `hash_insertar()` ya que no es el mismo tipo de función que la que se le pasa por parámetro a `hash_con_cada_clave()`

   Si pudo copiar todas las claves con éxito, entonces libera las claves, casilleros y listas del hash viejo. Hay que prestar mucha atención ya que no se tienen que liberar los valores que inserto el usuario. Para evitar esto, cambio el destructor del hash a NULL y luego llamo a la función de `liberar_tabla_hash()`. Finalmente, asigno el hash nuevo al hash viejo.

   Si no se pudieron copiar las claves, libero el hash nuevo pero nuevamente teniendo el cuidado de no liberar los elementos.

4. `hash_con_cada_clave()`
   A la hora de implementar esta función, use el iterador interno de lista. Sin embargo, el iterador de hash recibe un `bool (*funcion)(hash_t*, const char*, void*)` mientras que el de lista recibe un `bool (*funcion)(void*, void*)`. Por esta misma razón, tuve que hacer un wrapper llamado `hash_con_cada_clave_aux()`, al cual le paso una estructura `datos_funcion_t` que contiene todos los datos necesarios para aplicar la función que recibe `hash_con_cada_clave`.

   Como el iterador de lista itera mientras la función devuelva `true`, y el iterador de hash itera mientras la función devuelva `false`, el wrapper almacena el retorno de la función pasada por parámetro en `datos_funcion->retorno`. Este es el valor que define si se continua iterando en el iterador de hash. Luego, el wrapper devuelve `!(datos_funcion->retorno)` ya que es el valor utilizado para seguir iterando por la lista.

5. `obtener_posicion_casillero()`
   Esta es una función esencial para el TDA. A pesar de no ser una primitiva, es una función que me permite simplificar el código ya que la reuso en:
   - `hash_insertar()`
   - `hash_quitar()`
   - `hash_obtener()`
   - `hash_contiene()`
     Funciona usando el iterador externo de lista (para variar un poco y no tener que definir otra función auxiliar :D). Consiste en iterar sobre la lista ubicada en la posición de la clave hasheada hasta encontrar la clave o que no haya mas elementos por iterar. Aquí hay un diagrama que explica mejor el funcionamiento de la misma:

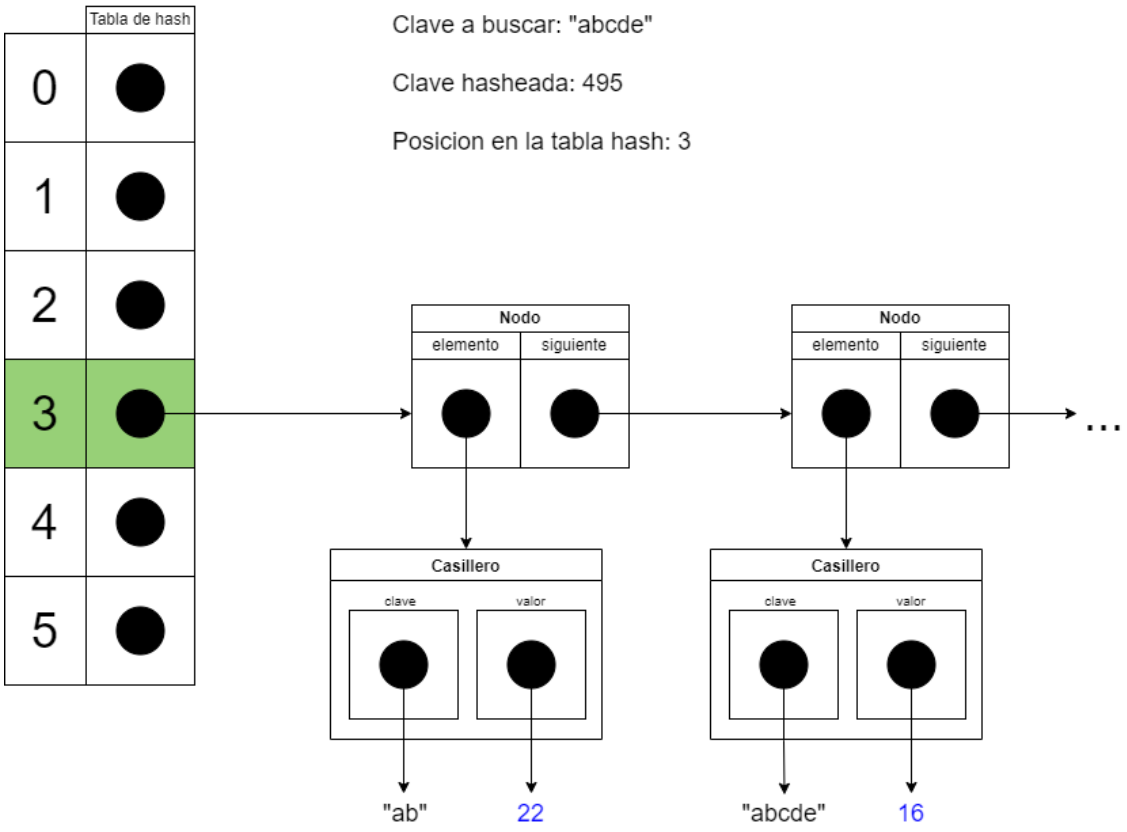


Tabla de hash
0
1
2
3
4
5
Clave a buscar: "abcde"
Clave hasheada: 495
Posicion en la tabla hash: 3
Nodo
elemento
siguiente
Nodo
elemento
siguiente
...
Casillero
clave
valor
Casillero
clave
valor
"ab"
22
"abcde"
16

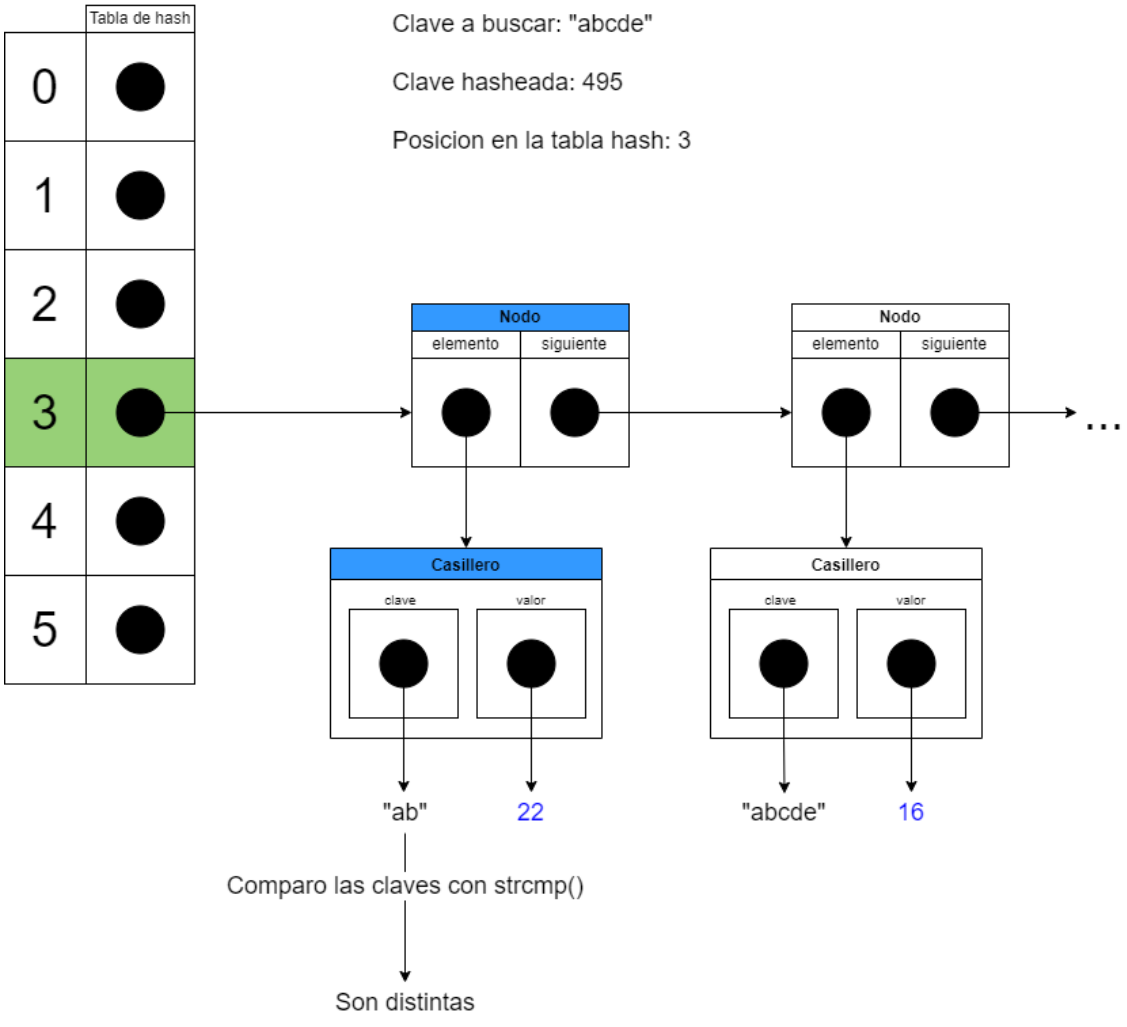


Tabla de hash
0
1
2
3
4
5
Clave a buscar: "abcde"
Clave hasheada: 495
Posicion en la tabla hash: 3
Nodo
elemento
siguiente
Nodo
elemento
siguiente
...
Casillero
clave
valor
Casillero
clave
valor
"ab"
22
"abcde"
16
Comparo las claves con strcmp()
Son distintas

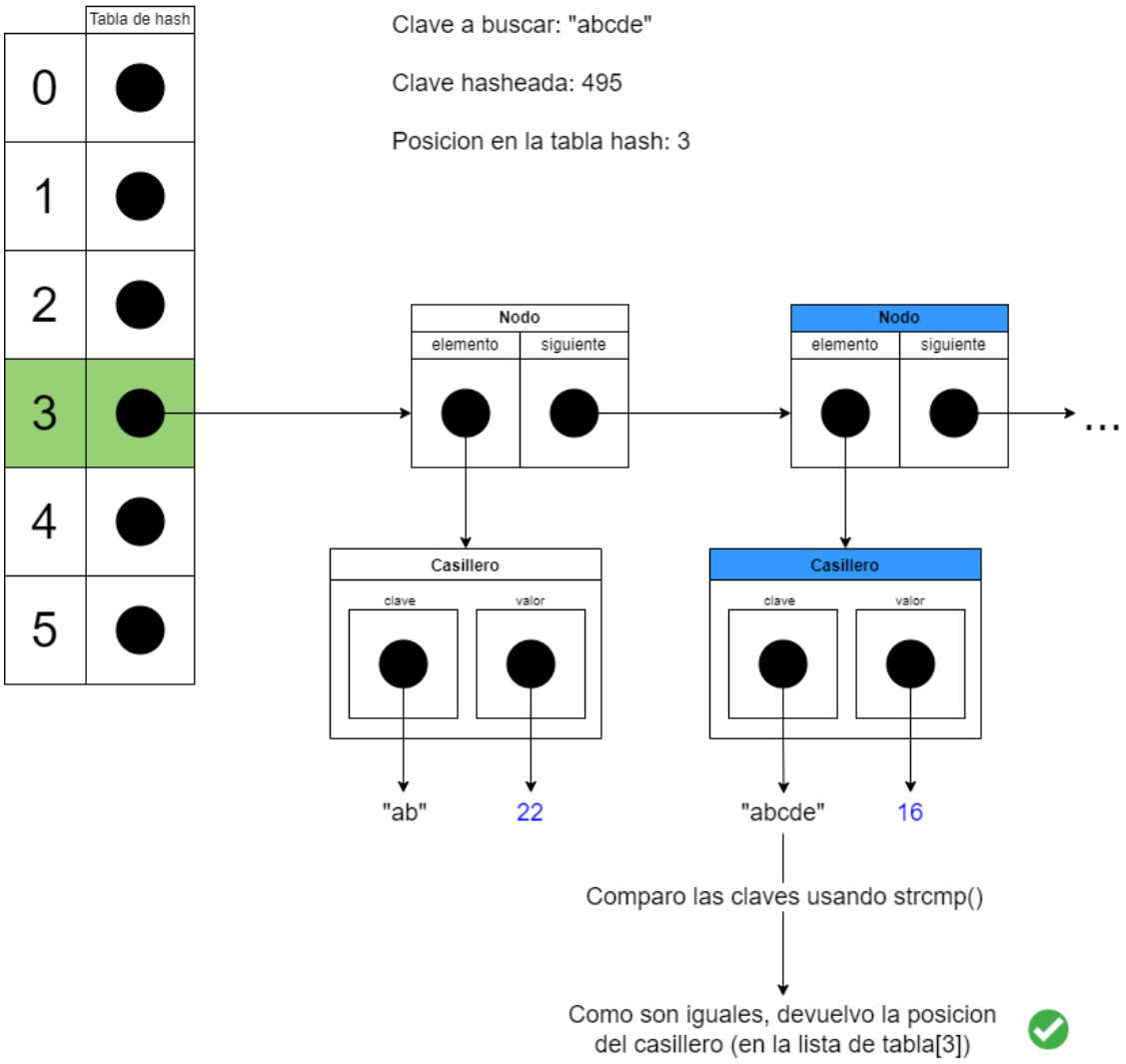


Tabla de hash
0
1
2
3
4
5
Clave a buscar: "abcde"
Clave hasheada: 495
Posicion en la tabla hash: 3
Nodo
elemento
siguiente
Nodo
elemento
siguiente
...
Casillero
clave
valor
Casillero
clave
valor
"ab"
22
"abcde"
16
Comparo las claves usando strcmp()
Como son iguales, devuelvo la posicion del casillero (en la lista de tabla[3])