

# Ayudantía 7



## B+ Trees & Hashing

2024-1

# MATERIAL DE APOYO

1. Cheatsheet C (notion resumen)
2. Ejercicios de práctica C
3. Cápsulas de semestres pasados

**Dónde encuentro esto?**

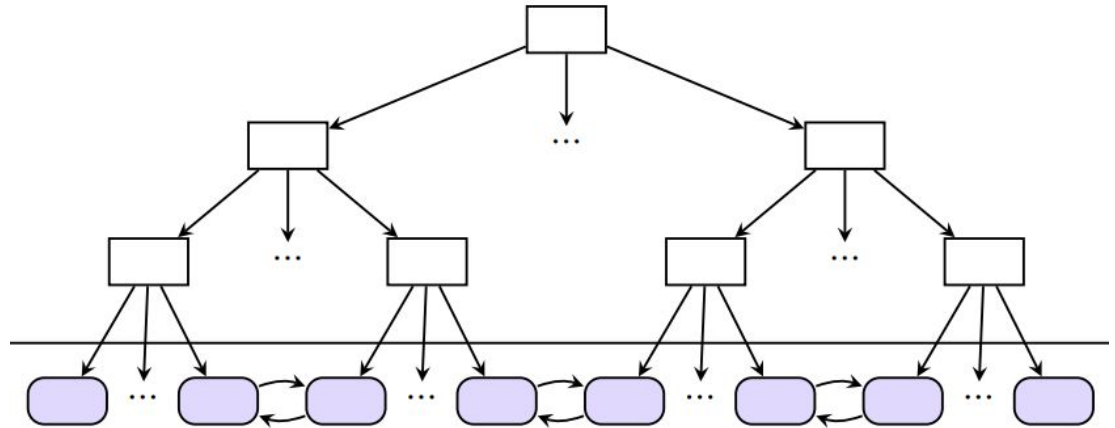
Links en ReadMe carpeta "Ayudantías" del repo



# B+ Tree

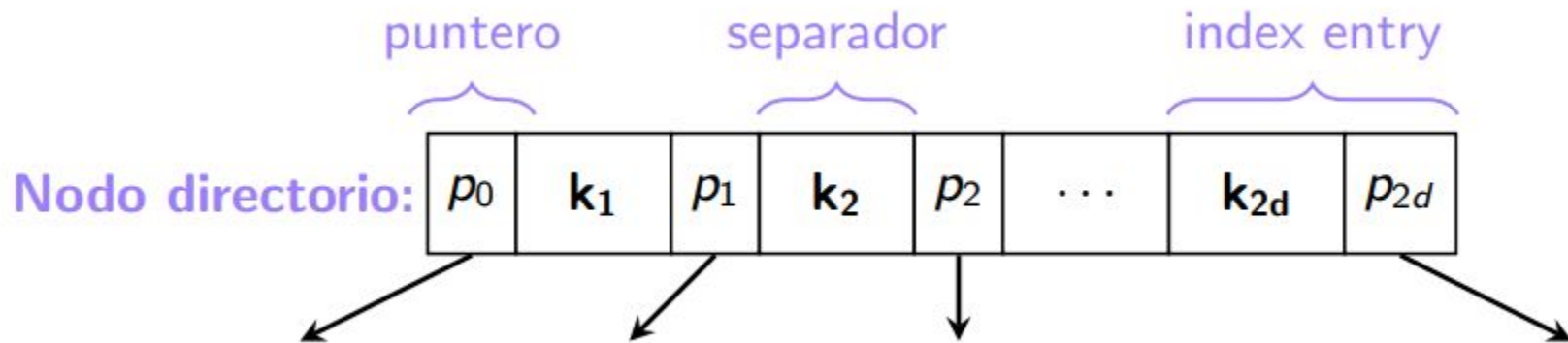
- Todas las hojas están al **mismo nivel** (altura)
- Solo las **hojas** tienen **valor**
- Las hojas están conectadas por una **lista doblemente ligada**
- Búsqueda, inserción y eliminación complejidad

$$O(\log_{2d}(n))$$



\*d: orden del árbol, que equivale al grado mínimo de los nodos internos

# Estructura de un nodo



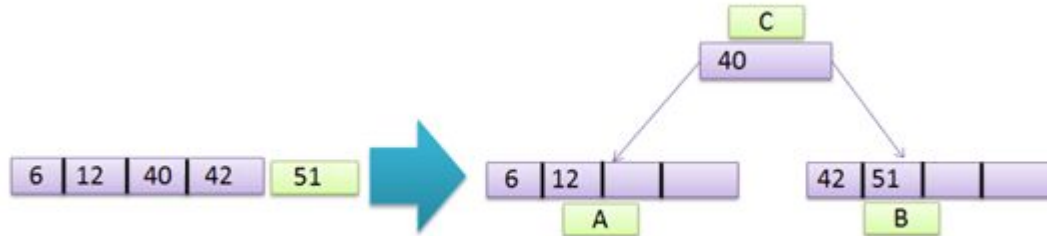
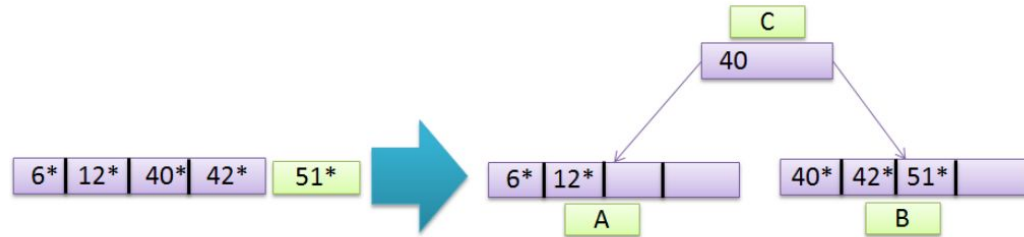
**Para orden  $d$ :**

	# Punteros (Hijos)	# Separadores
Raíz	$1 \leq n \leq 2d+1$	$1 \leq n \leq 2d$
Nodos internos	$d \leq n \leq 2d+1$	$d \leq n \leq 2d$

## Inserción

**InsertarEnBPlusTree**( $k$ , raiz):  
 $P \leftarrow \text{BusquedaEnArbol}(k, \text{raiz})$   
 $\text{Insertar}(k, P)$   
**if**  $\text{CantidadDeLlaves}(P) > 2d$  :  
     $\text{Split}(P)$

## 2 casos de Split ¿Cuál es la diferencia?

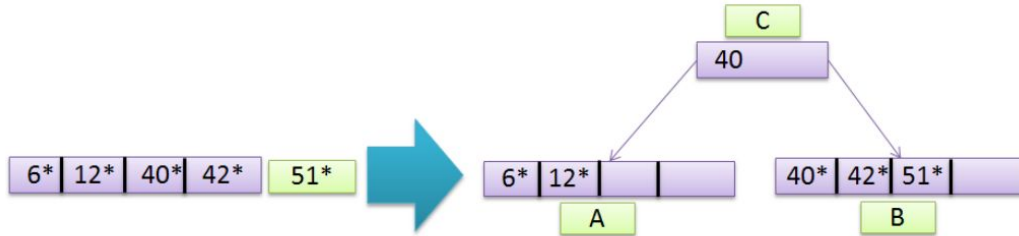


**d = 2**

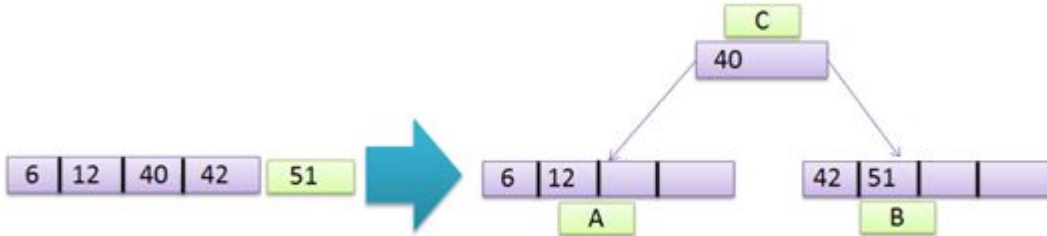
**Split**



**Separar las llaves en dos  
nodos distintos y  
reasignar punteros**



- Inserción en un **nodo hoja** lleno: 40\* se mantiene como dato y se copia en nodo padre como llave.



- Inserción en un **nodo interior** lleno: reacomodación de las llaves.

# ¡Veamos un ejemplo!

**Árbol de orden  $d = 1$**



- Partimos con un nodo con valor 5



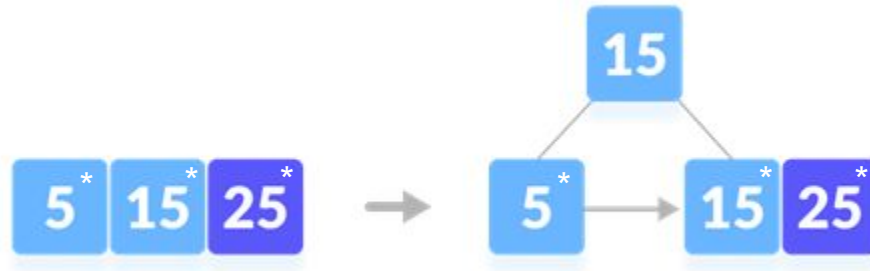
# Insert 15

**Árbol de orden  $d = 1$**



# Insert 25

Árbol de orden  $d = 1$



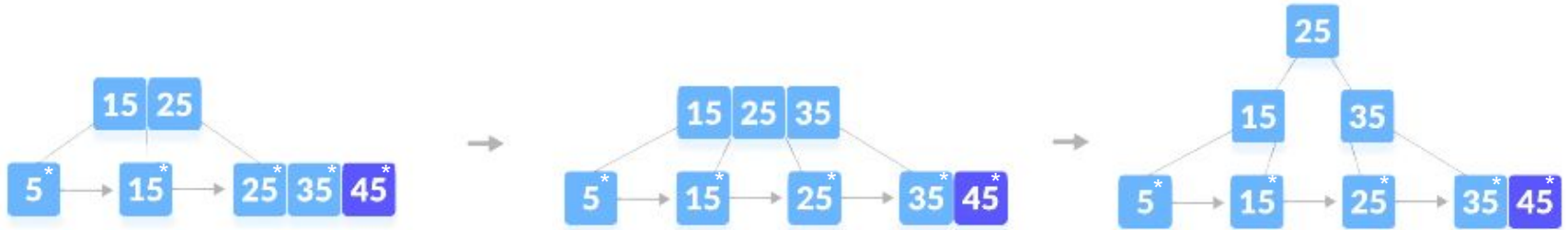
# Insert 35

Árbol de orden  $d = 1$



# Insert 45

Árbol de orden  $d = 1$



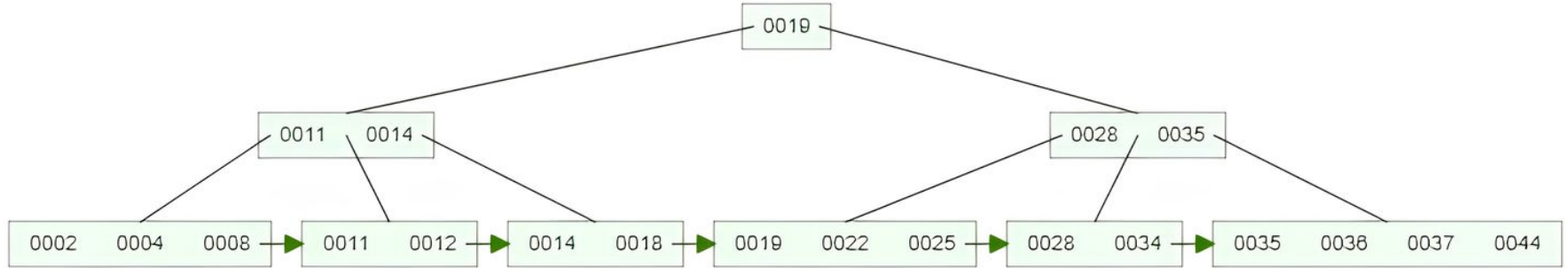
# Ejercicios

Inserte las siguientes llaves a un árbol B+:

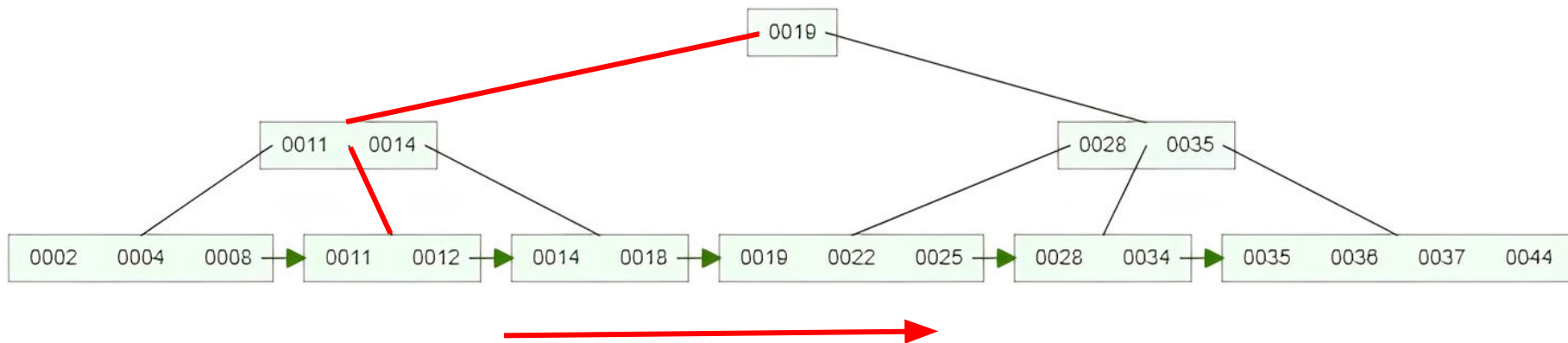
4, 12, 36, 28, 34, 22, 2, 44, 11, 8, 25, 35, 14, 18, 37, 19

Realice una búsqueda por rango entre 13 y 25

# Ejercicio



# Ejercicio



R: [14, 18, 19, 22, 25]

## Definición

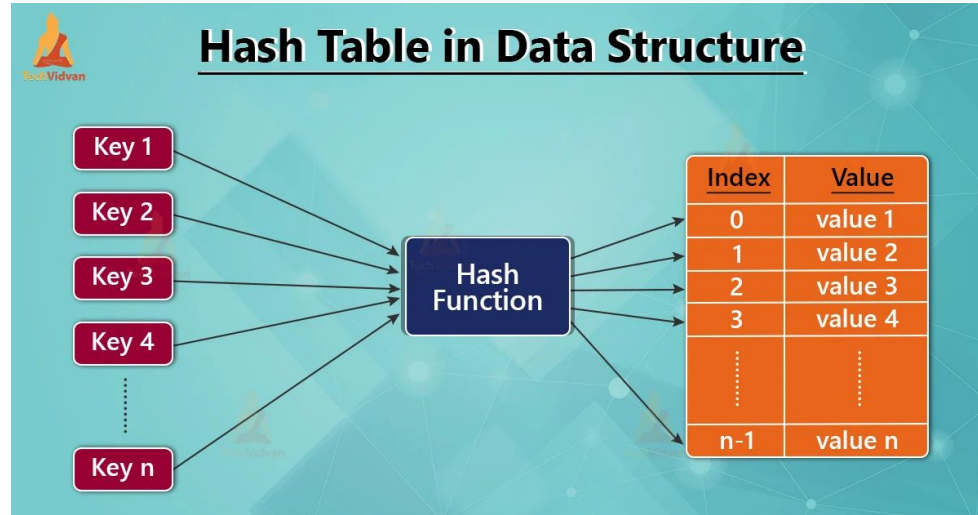
**Def:** Hashing es una manera de convertir información en una especie de "huella digital" única y fija que se puede utilizar para identificar o verificar esa información. Esto permite volver  $O(1)$  las búsquedas de datos.



# Tabla Hash

## Def:

- Es una estructura de datos que se usa para almacenar y recuperar datos almacenados mediante una búsqueda.
- Se hace uso de una función de hash para obtener un índice y agregarlo a la tabla.



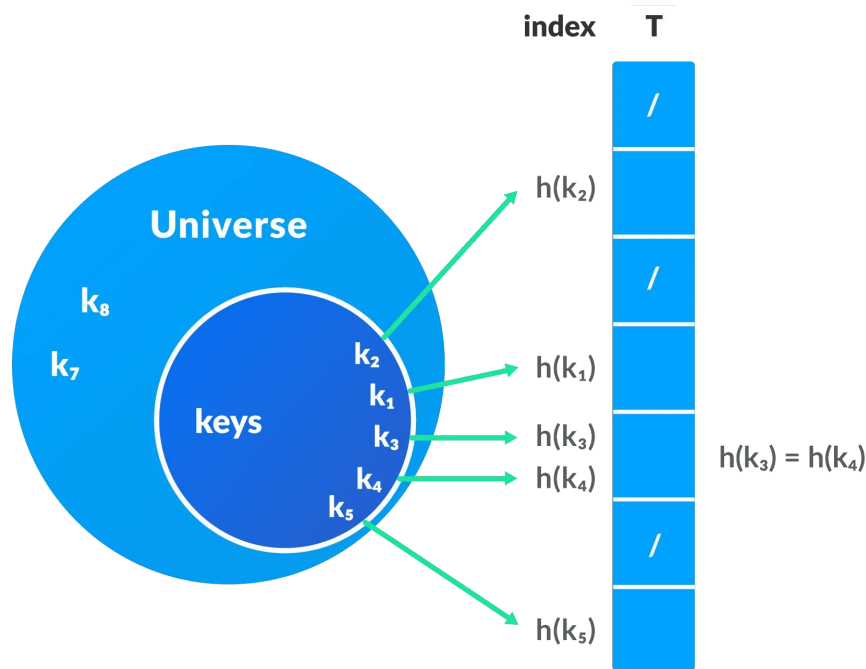
# Función de Hash

## Def:

- Se utilizan para convertir una clave en una posición en la tabla de hash.

## Ejemplo:

- Función de hash módulo:**  
 $\text{Hash}(\text{key}) = \text{key} \% \text{table\_size}$



# Función de Hash

## Ejemplo:

Digamos que tenemos los siguientes elementos: 1, 2, 3, 4

El tamaño de mi arreglo es 4, entonces:

$$1 \% 4 = 1$$

$$2 \% 4 = 2$$

$$3 \% 4 = 3$$

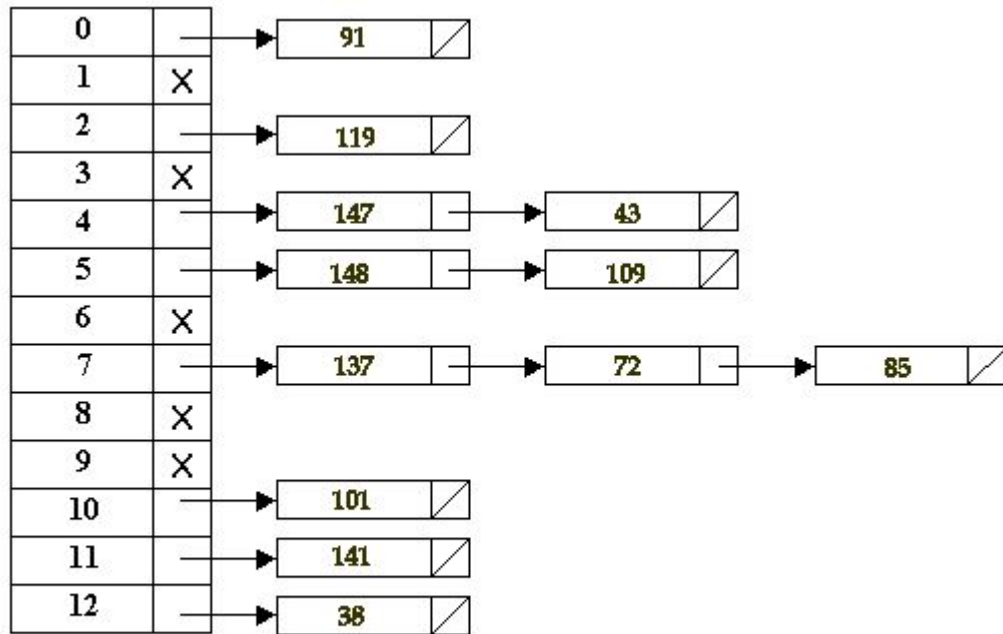
$$4 \% 4 = 0$$

Entonces, mi tabla hash es:

4	Index = 0
1	Index = 1
2	Index = 2
3	Index = 3

# Colisiones: Encadenamiento

- Una solución para las colisiones, es crear listas ligadas en las posiciones con conflicto.
- El problema de esto, es que la búsqueda pasa de ser  $O(1)$  a  $O(m)$ , siendo  $m$  la cantidad de elementos en la colisión.



Resultado usando LIFO.

# Colisiones: Encadenamiento

- Elementos: [1, 3, 4, 7, 9, 10, 12, 17, 19, 20]
- Tabla de hashing de 5 espacios

# Colisiones: Encadenamiento

- Elementos: [1, 3, 4, 7, 9, 10, 12, 17, 19, 20]
- Tabla de hashing de 5 espacios

$$1 \% 5 = 1$$

$$12 \% 5 = 2$$

$$3 \% 5 = 3$$

$$17 \% 5 = 2$$

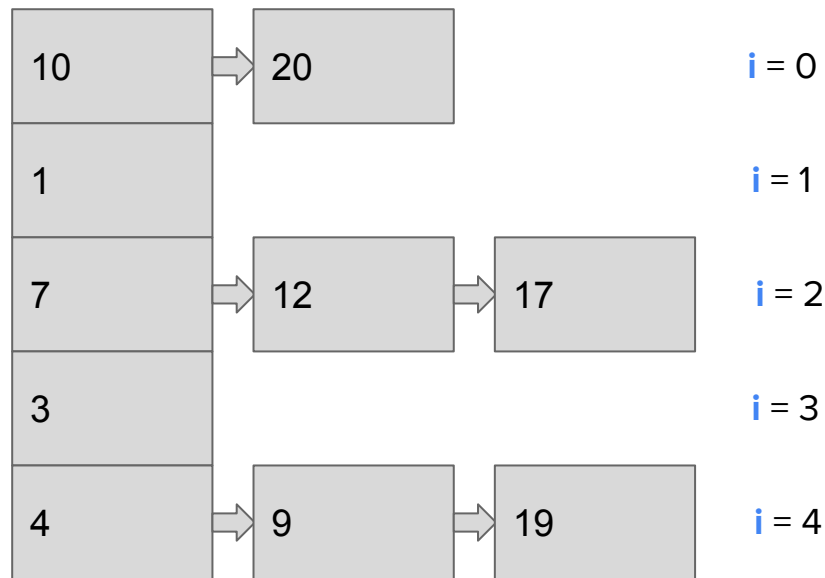
$$4 \% 5 = 4$$

$$19 \% 5 = 4$$

$$9 \% 5 = 4$$

$$20 \% 5 = 0$$

$$10 \% 5 = 0$$



# Colisiones: Por Exploración (Sondeo lineal)

- Aquí, cuando ocurre una colisión, seguimos avanzando por el arreglo hasta encontrar el primer espacio disponible.
- No les recomendamos usarlo, ya que termina provocando más problemas a la larga que soluciones.
- Ejemplo problema provocado:  
Eliminar llaves provoca problemas en la búsqueda.



# Ejemplo real UC (2023-1)

- En introducción a la programación, actualmente usamos la plataforma Clearn. El problema que teníamos, es que al dar prueba, era demasiado lenta para correr códigos.
- Al investigar, nos dimos cuenta de que el problema es que Clearn para correr el código, lo que hacía era **buscar linealmente ( $O(n)$ )** los datos del alumno, entre todos los alumnos. Suena poco, pero si tenemos en cuenta que una prueba de intro la dan en promedio **1000 personas**, todos corriendo su código al mismo tiempo, el sistema colapsa. En promedio, se demoraba **20 segundos** en correr el código de un alumno.
- **La solución, fue hashing.** Pasaron a todos los alumnos por una función hash, los pusieron a todos en una gran tabla hash, e hicieron que Clearn usara la función de hash para buscar en la tabla al alumno. **3 líneas en total.**
- Con eso, actualmente correr un código en Clearn se nos está demorando **menos de un milisegundo.**



# Ejercicio (2022 - 2, I2)

Suponga que en una ventana de tiempo recibimos varias palabras de  $\{a,b\}^*$ , y nos interesa contar cuántas veces recibimos cada una. Para esto, utilizaremos **la tabla de hash T de tamaño 7** que almacene las apariciones por palabra, y que usa la siguiente función de hash  $h \{a,b\}^* \rightarrow \{0, \dots, 6\}$ :

$$h(s_0s_1 \dots s_p) = \left( \sum_{i=0}^p \#(s_i) \right) \bmod 7$$

Considerando que  $\#(a) = 1$ , y  $\#(b) = 2$ , y considerando que T usa encadenamiento para resolver colisiones, ¿Cómo quedaría la tabla con los siguientes inputs?:

*aab, ab, bb, bab, aab, bab, abbaa*

## Ejercicio (2022 - 2, I2)

$$h(s_0s_1 \dots s_p) = \left( \sum_{i=0}^p \#(s_i) \right) \bmod 7$$

*aab, ab, bb, bab, aab, bab, abbaa*

$$4 \quad 3 \quad 4 \quad 5 \quad 4 \quad 5 \quad 7 \bmod 7 = 0$$

Considerando que  $\#(a) = 1$ , y  $\#(b) = 2$ , y considerando que T usa encadenamiento para resolver colisiones, ¿Cómo quedaría la tabla con los siguientes inputs?:

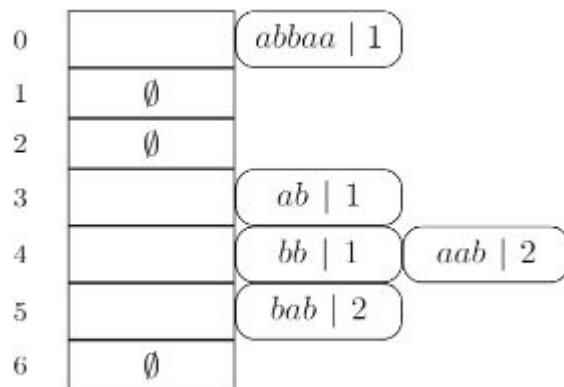
# Ejercicio (2022 - 2, I2)

$$h(s_0s_1 \dots s_p) = \left( \sum_{i=0}^p \#(s_i) \right) \bmod 7$$

*aab, ab, bb, bab, aab, bab, abbaa*

4      3      4      5      4      5      7 mod 7 = 0

Considerando que  $\#(a) = 1$ , y  $\#(b) = 2$ , y considerando que T usa encadenamiento para resolver colisiones, ¿Cómo quedaría la tabla con los siguientes inputs?:



## Ejercicio (Ayudantes)

- Tienes un arreglo de  $n$  números enteros ( $-n < x < n$ ), y quieres encontrar todas las duplas de números, que su suma de 0, y retornar el número a su valor absoluto.
- Input: [3, 4, -3, 3, 1, 3, 1, -4, -2, 2, -10]
- Output: [3, 4, 2]

## Ejercicio (Ayudantes)

- Input: [3, 4, -3, 3, 1, 3, 1, -4, -2, 2, -10]
- Output: [3, 4, 2]

### Primera idea: Fuerza bruta

En cada elemento, revisar el arreglo entero por si existe su elemento neutro. Por lo que recorreríamos el arreglo  $n$  veces, y en cada vez, volveríamos a recorrerlo  $n$  veces. Dando esto  $O(n^2)$ .

## Ejercicio (Ayudantes)

- Input: [3, 4, -3, 3, 1, 3, 1, -4, -2, 2, -10]
- Output: [3, 4, 2]

### Segunda idea: **tabla hash**

- Creamos un **arreglo A de tamaño n**, con todos los valores iniciales 0. Recorremos una vez el arreglo, añadiendo los elementos con la siguiente condición. **Si  $x \leq 0$ ,  $A[-x] = 1$ .**
- Volvemos a recorrer el arreglo. Si  $x > 0$ , revisamos  $A[x]$ . Si es 1, añadimos x a nuestro output.
- Como recorremos el arreglo solo 2 veces, tenemos  $O(n)$  en eficiencia.

# Demo: Hashing en C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string.h>

#define TABLE_SIZE 10

typedef struct Entry {
    int key;
    int value;
    struct Entry* next;
} Entry;

Entry* newEntry(int key, int value) {
    Entry* new_entry = calloc(1, sizeof(Entry));
    new_entry->key = key;
    new_entry->value = value;
    new_entry->next = NULL;
    return new_entry;
}
```


# Demo: Hashing en C



```
int hash_function(int key) {  
    return key % TABLE_SIZE;  
}  
  
void insert(Entry** table, int key, int value) {  
  
    Entry* new_entry = newEntry(key, value);  
  
    int index = hash_function(new_entry -> key);  
  
    Entry* current = table[index];  
    if (current == NULL) {  
        table[index] = new_entry;  
    } else {  
        while (current->next != NULL) {  
            current = current->next;  
        }  
        current->next = new_entry;  
    }  
}
```



# Demo: Hashing en C




```
Entry* search(Entry** table, int key, int value) {
    int index = hash_function(key);
    Entry* current = table[index];
    while (current != NULL) {
        if (current->key == key && current->value == value) {
            printf("Found a node in the table at index %i \n", index);
            return current;
        }
        current = current->next;
    }
    printf("Did not find a node in the table at index %i \n", index);
    return NULL;
}
```

# Demo: Hashing en C

```
void delete(Entry** table, int key, int value) {  
    int index = hash_function(key);  
    Entry* current = table[index];  
    Entry* prev = NULL;  
    while (current != NULL) {  
        if (current->key == key && current->value == value) {  
            if (prev == NULL) {  
                // The element to delete is the first in the list  
                table[index] = current->next;  
            } else {  
                // The element to delete is in the middle or end of the  
list  
                prev->next = current->next;  
            }  
            free(current);  
            return;  
        }  
        prev = current;  
        current = current->next;  
    }  
}
```

## Demo: Hashing en C



```
void print_collisions_at_index(Entry** table, int index) {
    Entry* current = table[index];
    if (current == NULL) {
        printf("No collisions at index %d\n", index);
        return;
    }
    printf("Collisions at index %d:\n", index);
    while (current != NULL) {
        printf("Key: %d, Value: %d\n", current->key, current->value);
        current = current->next;
    }
}
```

# Demo: Hashing en C

```
void free_table(Entry** table){  
    for(int i = 0; i < TABLE_SIZE; i ++){  
        Entry* current = table[i];  
        while(current != NULL){  
            Entry* temp = current;  
            free(current);  
            current = temp->next;  
        }  
    }  
    free(table);  
}
```