Hashing & Rojo Negro

2025-1

Hashing

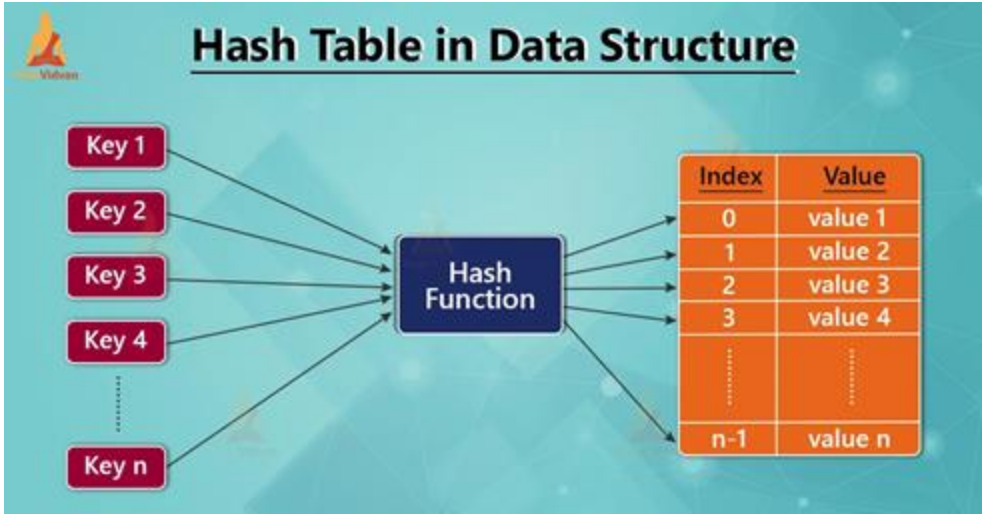
HASHING

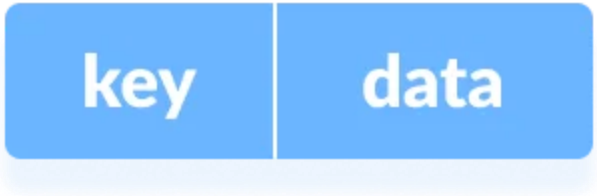
Def: Hashing es una manera de convertir información en una especie de "huella digital" única y fija que se puede utilizar para identificar o verificar esa información. Esto permite volver O(1) las búsquedas de datos.

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/buscarlo en la tabla.





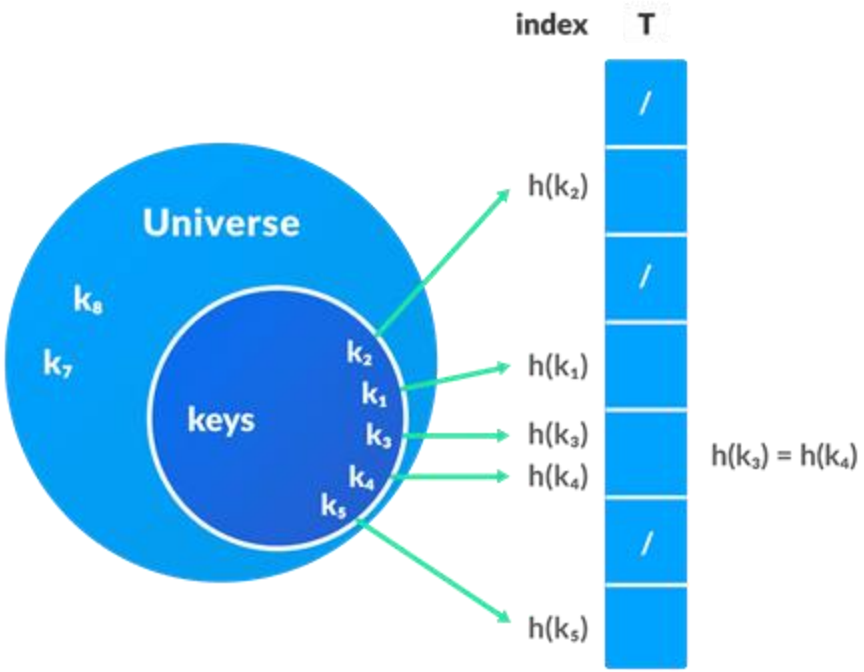
Función de Hash

Def:

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

Ejemplo:

- **Función de hash módulo:** Hash(key) = key % table_size



Función de Hash

Ejemplo:

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

El tamaño de mi arreglo es 4, entonces:

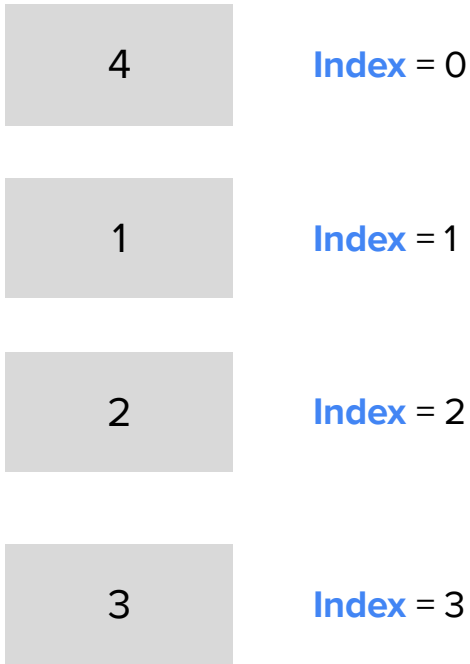
1 % 4 = 1

2 % 4 = 2

3 % 4 = 3

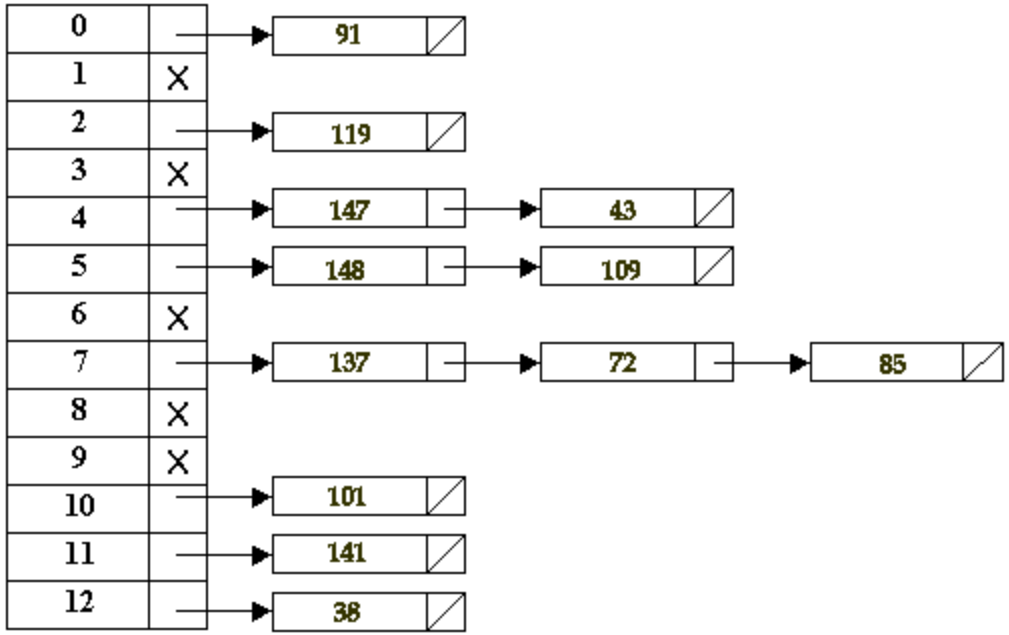
4 % 4 = 0

Entonces, mi tabla hash es:



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: 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.

Demo: Hashing en C

```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

```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

```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

```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

```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

```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);

}
```

Ejercicio 1 (I2 2024-1)

Considere el problema de entregar los resultados de la prueba PAES del proceso de admisión a las universidades 2025. Este proceso es de muy alta carga en un período muy corto de tiempo. Ud. recibirá del DEMRE el archivo con los resultados solo 2 horas antes de la hora de publicación: 00:00 del día 20 de enero. Ud. considera que el servidor de bases de datos no sería capaz de soportar la carga por lo que considera usar una Tabla de Hash para almacenar los resultados, la cual sería poblada (Insert) entre las 10:00 y 11:59 del día previo (19 de enero).

Los datos están almacenados en un registro con los siguientes campos:

Record r(
RUN Entero (sin DV)
DV char
Nombre String 40
NEM Entero
Ranking Entero
Puntaje competencia lectora Entero
Puntaje competencia matemática 1 Entero
Puntaje historia y ciencias sociales Entero
Puntaje ciencias Entero
Puntaje competencia matemática 2 Entero
)

Considere un rango de los RUN 1-25.000.000, pero solo 200 mil personas rinden la prueba PAES y el 80 % tienen un RUN entre 22.000.000 y 22.400.000

a) Suponiendo que el registro puede almacenarse directamente en una tabla de Hash (no es necesario el puntero al registro). Proponga una estructura de datos para dicha Tabla, indicando la función de hash utilizada, el tamaño de la tabla y su factor de carga esperado.

Considere un rango de los RUN 1-25.000.000, pero solo 200 mil personas rinden la prueba PAES y el 80 % tienen un RUN entre 22.000.000 y 22.400.000

a) Suponiendo que el registro puede almacenarse directamente en una tabla de Hash (no es necesario el puntero al registro). Proponga una estructura de datos para dicha Tabla, indicando la función de hash utilizada, el tamaño de la tabla y su factor de carga esperado.

Respuesta: Dado que el dominio del RUN es de 1 a 25.000.000, pero solamente rinden la prueba PAES 200.000 personas, y además el **80 % de los registros** se encuentran entre **RUN 22.000.000 y 22.400.000**, se puede aprovechar esta distribución para diseñar una tabla hash eficiente.

Tamaño de la tabla: 200.000 buckets

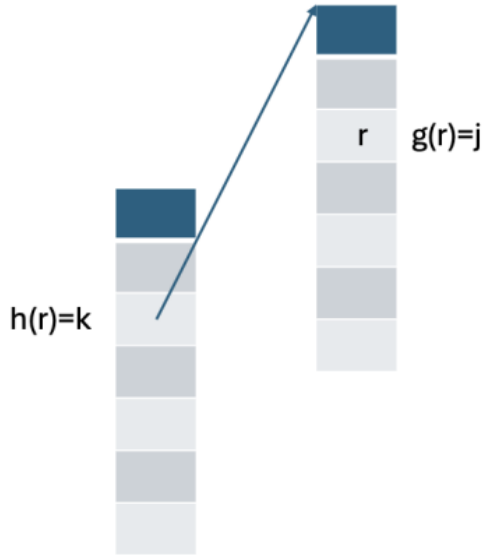
Función de hash:

$$\mathrm{H(RUN)} = \frac{RUN\ - 22.000.000}{2}$$

Factor de carga 100 por ciento. También es válido usar funciones con más buckets. Como el número de registros es fijo y conocido no se necesita mas espacio para nuevas inserciones.

b) ¿Cómo cambia su solución si cada bucket de la primera tabla de hash contiene solo un puntero a una segunda tabla de hash que almacena los resultados?

b) ¿Cómo cambia su solución si cada bucket de la primera tabla de hash contiene solo un puntero a una segunda tabla de hash que almacena los resultados?
Respuesta: h(RUN) =DV genera una tabla de 11 valores, g(RUN) RUN mod 7 pueden ser otras funciones.



c) Para el segundo caso, proponga el pseudocódigo de las funciones HashInsert(A,k,j,r) y HashSearch (A,k,j) → r. Siendo A la tabla de Hash, h(r)= k el resultado de la función de hash de la primera tabla y g(r)=j la de la segunda y r el registro conteniendo los resultados de una persona.

```
k <- h(r)
j <- g(r)
Insert(A,k,j,r) {

    if A[k] = null  A[k] <- alloc(buckets) --buckets=tamaño de tabla secundaria
    else return false
    if ( A[k][j] está vacío) <- r; return true
    else j <-overflow (A[k],j); <- r; return true
    return false
 }
 HashSearch(A,k,j) {
    if A[k]= null return null
    j <- overflow A[k][j]
    if A[k][j]= vacío return null
    else return A[k][j]
}
overflow (p){
    while true
    if no hay overflow return p
    else r <- busque en el overflow     -- el overflow depende de la técnica elegida
}
```

Árboles Rojo Negro

Árboles Rojo-Negro

Es un **tipo** de **ABB** que cumple las siguientes propiedades:

1. Cada nodo es **rojo** o **negro**

2. La **raíz** del árbol es **negra**

3. Si un nodo es **rojo**, entonces sus hijos deben ser **negros**

4. La cantidad de nodos **negros** en el camino a cada hoja debe ser la misma (esta es la noción del balance)

Extra.a: Los nodos nulos/hojas vacías se consideran **negros**

Extra.b: Las inserciones son con nodos **rojos**

Ejercicio

3. Respecto a los árboles rojo negro:

   a) Justifica que la rama más larga del árbol tiene a lo más el doble de nodos que la rama más corta. Entiéndase por rama la ruta de la raíz hasta una hoja.

   b) En el algoritmo de inserción estudiado en clases, un nodo recién insertado se pinta de rojo. Con esto, corremos el riesgo de violar la propiedad 3 de árbol rojo-negro (según las diapositivas); y cuando así ocurre, usamos rotaciones y cambios de color para restaurar esa propiedad. En cambio, si pintásemos el nodo de negro, no correríamos este riesgo.
      i) ¿Por qué no pintamos de negro un nodo recién insertado?

Parte A

La cantidad de nodos negros en la rama más corta debe ser la misma que en la más larga (por propiedad 4). Llamemos a esta cantidad “**n**".

La rama más corta posible tendrá todos los nodos negros (será de **n** nodos)
Ahora, **la rama más larga posible será de n + r nodos**, donde r es la cantidad máxima de nodos rojos que puede tener.

Para analizar el máximo r, consideremos la propiedad 3, con la que sabemos que si un nodo es rojo, sus hijos deben ser negros (hojas nulas se consideran negros).

Si consideramos además que la raíz es negra, sabemos que la cantidad de nodos rojos será menor a la de negros (por cada rojo en una rama estará su hijo negro ($(n - 1) \geq r$). Es decir, la rama más larga posible tendrá n + r nodos donde r = n-1. **Entonces tendrá 2n - 1 nodos**, el doble de los n nodos de la más corta posible. Entonces, podemos decir que la rama más larga de un ARN tendrá a lo más el doble de nodos que la más corta.

Parte B

Porque para que un árbol sea RN, debe cumplir con la propiedad 4. Si insertamos una hoja negra a un árbol que ya es RN y queda con más de una rama, siempre romperemos la propiedad (excepto para la raíz)

Esto se explica ya que, si antes tenía “n” nodos negros en cada rama, ahora tendrá “n + 1” en una de ellas, y “n” en las demás. Así, al insertar el nodo como negro la propiedad se rompe siempre, mientras que si insertamos el nodo como rojo, romperá la propiedad 3 en algunos casos.

Además, restaurar la propiedad 4 será en general más costoso, ya que incide globalmente en el árbol, mientras que la propiedad 3 es más fácil de arreglar con cambios locales.