

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

El Tipo de Dato Abstracto Diccionario

- El tipo de dato abstracto diccionario consiste de un conjunto de elementos y las operaciones:
- insertar, eliminar, buscar un elemento.

Utilizaremos una estructura de datos para implementar un Diccionario que llamaremos Tabla de Hash.

Tabla de Hash

- Una tabla de hash es un tipo de datos (concreto) que permite almacenar un **conjunto** de objetos.
- Es una estructura eficiente que permite **insertar, eliminar y buscar** un elemento en un **tiempo promedio “CONSTANTE”**
- Es una estructura muy utilizada en computación. Por ejemplo, los compiladores de lenguajes las utilizan para almacenar la tabla de símbolos del programa (variables, etc.)

El Tipo de Dato Tabla de Hash

- Las únicas operaciones permitidas son:
 - insertar(Tipo x): agregar el elemento x
 - eliminar(Tipo x): eliminar el elemento x
 - buscar(Tipo x): devuelve verdad si está el elemento x
 - vaciar(): convertir en vacía la tabla de hash
 - es_vacia(): devuelve verdad si la tabla está vacía
 - num_elem(): devuelve el número de elementos en la Tabla

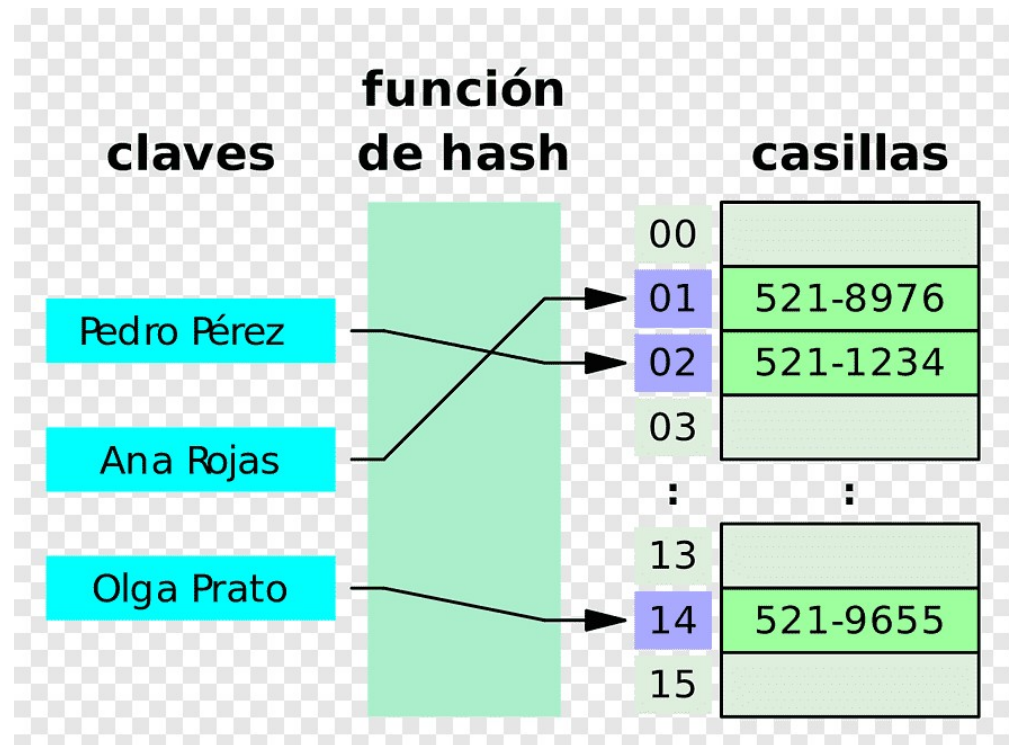
- Una tabla de Hash **consiste de un arreglo de tamaño dado (Capacidad)**, cuyos elementos estarán almacenados en las posiciones **0** a **Capacidad-1**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Capacidad = 10

- **El universo de posibles objetos normalmente es mucho mayor que la tabla** y no tiene sentido definir una tabla con tantos elementos como el universo, si el número de elementos que puede ocurrir es mucho menor.
- **Los elementos poseen una clave única que los identifica (por ejemplo, número cédula de identidad)**, y es mediante esa clave que se determina la posición del elemento en la tabla.

- Para ello se utiliza una **función de hash** que convierte la clave en un número (**hashcode**) y al aplicarle módulo **Capacidad** nos da una posición de la tabla de **0 a Capacidad-1** donde insertar el elemento.
- Una clave puede ser un número o una cadena de caracteres.
- Lo ideal es que la función de hash permita asignar un objeto en una posición distinta de otros. Pero esto es imposible si el universo de objetos es mayor que el tamaño de la tabla



Situación ideal:

john se ubica mediante hash en la posición 3,

phil se ubica mediante hash en la posición 4

dave en la posición 6

mary en la posición 7

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Situación normal:

Puede haber **colisiones**, dos objetos al aplicarle la **función de hash a sus claves, da el mismo valor**

El rendimiento entonces de una tabla de hash dependerá de la **función de hash y de la estrategia para resolver colisiones (elementos con claves iguales).**

Normalmente el **tamaño de la tabla es un número primo (porque se usa la función MOD (%) para determinar la posición del elemento en la tabla y un número primo permite mayor dispersión de las claves)**

Ej: Si el tamaño de la tabla es 12 los múltiplos de 4 (divisor de 12) solo podrán caer en la posiciones 0, 4 y 8 de la tabla.

Si el tamaño de la tabla es 11 (un número primo), un múltiplo de 4 podrá caer en cualquiera de las 11 posiciones.

Veamos una mala función de hash:

El tamaño de la tabla es 10.007 (número primo)

La clave de los elementos es una palabra en inglés: Queremos almacenar un diccionario en una tabla de hash.

El número de letras del inglés es 26 más el caracter en blanco
= 27 caracteres

key es un string que contiene la clave. Tomemos como hashCode:

$$\text{key}[0] + 27 * \text{key}[1] + 27^2 * \text{key}[2]$$

Por ejemplo, si la palabra es **year**: 'y' es 121, 'e' es 101, 'a' es 97, 'r' es 114. (Tomamos el código ASCII del caracter)

$$\text{hashCode de "year"} \text{ es } 121 + 27 * 101 + 27^2 * 97 = 73.561$$

$$\text{Al almacenarlo en la tabla, correspondería la posición } 73.561 \% 10.007 = 3.512$$

El número de Strings de tres letras es $26^3 = 17,576$

Un estudio sobre un diccionario de Inglés revela que el número de combinaciones de las primeras tres letras de una palabra es 2,851.

Por lo que si suponemos que cada secuencia de tres letras produce un valor distinto de hash, sólo el $2.851/10.007 = 28,5\%$ de las posiciones de la tabla se pueden obtener con esa función de hash

Por lo que la función de hash no es apropiada para el tamaño de la tabla pues habrá muchas colisiones (palabras con las mismas primeras tres letras) y el resto de posiciones (10.007-2851) no se pueden obtener

Una función de hash que se considera suficientemente buena. **El hashCode:**

$$\sum_{i=0}^{KeySize-1} Key[i] \cdot 37^i$$

Evaluar un polinomio en 37 (número primo). Normalmente no se toman todos los caracteres de la clave y se evalúa mediante la regla de Horner (puede haber overflow, no importa C++ lo permite, da un valor... que puede ser negativo)

$$h_k = k_0 + 37k_1 + 37^2k_2 = ((k_2) * 37 + k_1) * 37 + k_0$$

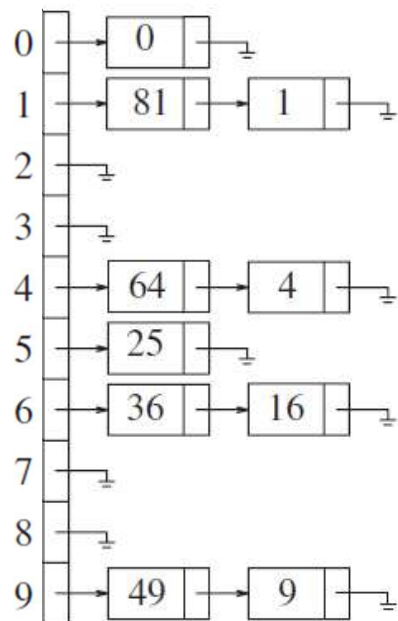
Y luego determinar la posición en la tabla con % Capacidad

Veamos como se resuelven las colisiones.

Hay dos tipos de implementaciones de tablas hash:

- 1) **El hashing abierto (open hashing) o encadenamiento separado (separate chaining):** se va creando una lista con todos los elementos con iguales posiciones (con igual valor del módulo del hashcode).
- 2) **El hashing cerrado (closed hashing) o direccionamiento abierto (open addressing):** todos los elementos se almacenan en la tabla pero cuando ocurre una colisión se debe tener una estrategia para reubicar los elementos. El término “open addressing” se debe precisamente a este hecho.

Veamos en que consiste el hashing abierto (separate chaining): se va creando una lista de todos los elementos con igual valor en la función de hash MOD tamaño. Suponga que la tabla tiene **capacidad 10** y se van agregando los elementos en el siguiente orden: 81,0,1,36,64,4,25,49,9,16



$\text{hashCode}(x) = x$

Posición en la tabla = $x \bmod 10$

El tamaño de la tabla es 10 (aunque mejor sería un número primo...)

Al agregar 1 surge una colisión, pues 1 y 81 tienen la misma posición

Entonces 1 se guarda al final (o al comienzo) de la misma lista donde está 81

Construyamos juntos la tabla completa,
insertando los elementos de último en la lista
Para una tabla con 7 elementos y los valores a
insertar son: 81,0,1,36,64,4,25

81,0,1,36,64,4,25

El **factor de carga** es el cociente: (número de elementos en la tabla)/(capacidad de la tabla)

Si la función de hash es lo suficientemente buena como para distribuir uniformemente los elementos en la tabla y el factor de carga está cerca de 1 **las operaciones serían, en principio, constantes.**

Si el factor de carga aumenta se puede aumentar el capacidad de la tabla y reubicar los elementos en una nueva tabla que reemplaza a la anterior.

Las claves de los objetos de la tabla deberían poder compararse por igualdad (si dos elementos son iguales o no) y tener **una función “hashCode”** que calcula la función de HASH, el valor al que se aplicará “% Capacidad” para obtener el índice en la tabla

Veamos una posible implementación de la tabla de hash abierto.....

```
#include <iostream>
#include <string>
#include <vector>
using std::string, std::vector;
// La Tabla sera un vector de apuntadores a nodo
struct nodo {
    // clave es un string
    string clave;
    // valor es un string, información adicional
    string valor;
    struct nodo* proximo;
};
// Inicializar un nodo
void inicializarNodo(nodo& nodo,
                    string clave, string valor) {
    nodo.clave = clave;
    nodo.valor = valor;
    nodo.proximo = nullptr;
}
```

```
struct TablaHash {
    // numero de elementos y capacidad de la tabla
    int num_elem, capacidad;
    // vector de apuntadores a nodo
    vector<nodo*> arr;
};

// Inicializar la tabla
void inicializarTabla(TablaHash& mp, int capacidad) {
    // capacidad de la tabla
    mp.capacidad = capacidad;
    mp.num_elem = 0;
    mp.arr = vector<nodo*>(mp.capacidad);
}
```

```
int posicion(TablaHash& mp, string clave) {
    // devuelve de una vez el índice en la tabla
    long long suma = 0, factor = 1;
    for (int i = 0; i < clave.size(); i++) {
        // suma = suma + (valor ascii de
        // char * (primo ^ i))...
        // donde i = 0, 1, 2, 3....n
        suma = (suma +
            (((int)clave[i]) * factor) % mp.capacidad)
            % mp.capacidad;
        factor = (factor * 37) ;
    }
    // si por overflow da negativo, ponerlo positivo
    if (suma < 0) {
        suma += mp.capacidad;
    }
    return suma;
}
```

```
void insertar(TablaHash& mp, string clave, string valor) {

    // Se busca el indice donde ira clave-valor
    int indiceCasilla = posicion(mp, clave);
    // se crea e inicializa un nodo
    nodo* nuevoNodo = new nodo;
    inicializarNodo((*nuevoNodo), clave, valor);
    // la casilla esta vacia, no hay colision
    if (mp.arr[indiceCasilla] == nullptr)
        mp.arr[indiceCasilla] = nuevoNodo;
    // Colision
    else {
        // agregar nodo de primero en la lista
        nuevoNodo->proximo = mp.arr[indiceCasilla];
        mp.arr[indiceCasilla] = nuevoNodo;
    }
    mp.num_elem++;
}
```

```
void eliminar(TablaHash& mp, string clave) {
    // obtener indice casilla para la clave dada
    int indiceCasilla = posicion(mp, clave);
    nodo* anterior = nullptr;
    // apuntar a la cabeza de la lista apuntada
    //por la casilla
    nodo* nodoActual = mp.arr[indiceCasilla];
```

```
while (nodoActual != nullptr) {
    // borrar nodo si coincide su clave
    if (clave == nodoActual -> clave) {
        // Borrar la cabeza de la lista
        if (nodoActual == mp.arr[indiceCasilla])
            mp.arr[indiceCasilla] = nodoActual->proximo;
        // Borrar otro nodo
        else
            anterior->proximo = nodoActual->proximo;
        delete nodoActual;
        mp.num_elem--;
        break;
    }
    anterior = nodoActual;
    nodoActual = nodoActual->proximo;
}
return;
```

```
string buscar(TablaHash& mp, string clave) {
    // Devuelve valor asociado a clave
    int indiceCasilla = posicion(mp, clave);
    // Apunta a la cabeza de la lista
    nodo* cabezaCasilla = mp.arr[indiceCasilla];

    while (cabezaCasilla != nullptr) {

        // Si la clave esta
        if (cabezaCasilla->clave == clave)
            return cabezaCasilla->valor;
        cabezaCasilla = cabezaCasilla->proximo;
    }
    // Si la clave no se consigue
    return "No se encontro";
}
```

```
int main() // prueba de la tabla hash
{
    // Inicializar la Tabla mp
    TablaHash mp ;
    inicializarTabla(mp,2);
    insertar(mp, "3456", "Carlos Perez");
    insertar(mp, "2391", "Maria Hernandez");
    insertar(mp, "3345", "Oscar Delgado");
    std::cout << buscar(mp, "2391") << std::endl;
    std::cout << buscar(mp, "3456") << std::endl;
    std::cout << buscar(mp, "3345") << std::endl;
    // clave no existe
    std::cout << buscar(mp, "9999") << std::endl;
    // Borrar nodo con clave dada
    eliminar(mp, "3456");
    // Buscar clave borrada
    std::cout << buscar(mp, "3456") << std::endl;
    return 0;
}
```


- Faltaron las funciones:
- vaciar(): convertir en vacía una tabla hash
- es_vacia(): devuelve verdad si la tabla está vacía
- num_elem(): devuelve el número de elementos en la Tabla
- imprimir(): imprime los elementos en la tabla
- Note que el código anterior puede insertar dos elementos con la misma clave, lo cual no es correcto. ¿Qué habría que hacer?
- Veamos la próxima transparencia.....

```
void insertar(TablaHash& mp, string clave, string valor) {
    if (buscar(mp,clave) != "No se encontro") return;
    // Se busca el indice donde ira clave valor
    int indiceCasilla = posicion(mp, clave);
    // se crea e inicializa un nodo
    nodo* nuevoNodo = new nodo;
    inicializarNodo((*nuevoNodo), clave, valor);
    // la casilla esta vacia, no hay colision
    if (mp.arr[indiceCasilla] == nullptr)
        mp.arr[indiceCasilla] = nuevoNodo;
    // Colision
    else {
        // agregar nodo de primeo en la lista
        nuevoNodo->proximo = mp.arr[indiceCasilla];
        mp.arr[indiceCasilla] = nuevoNodo;
    }
    mp.num_elem++;
}
```

Deberíamos tener una función llamada **rehash** que al aumentar el factor de carga mayor que 0.6 (recomendado) al momento de insertar un elemento, duplica el vector y reinserta los elementos de la vieja tabla en la nueva tabla aumentada, y elimina la vieja tabla.

¿Cómo haríamos eso en la implementación que vimos?

Pensémoslo un momento...



```
void insertar(TablaHash& mp, string clave, string valor) {
    if (buscar(mp,clave) != "No se encontro") return;
    if ((tabla.num_elem+1)*(1.0)/tabla.capacidad > 0.6) rehash(tabla);
    // Se busca el indice donde ira clave valor
    int indiceCasilla = posicion(mp, clave);
    // se crea e inicializa un nodo
    nodo* nuevoNodo = new nodo;
    inicializarNodo((*nuevoNodo), clave, valor);
    // la casilla esta vacia, no hay colision
    if (mp.arr[indiceCasilla] == nullptr)
        mp.arr[indiceCasilla] = nuevoNodo;
    // Colision
    else {
        // agregar nodo de primeo en la lista
        nuevoNodo->proximo = mp.arr[indiceCasilla];
        mp.arr[indiceCasilla] = nuevoNodo;
    }
    mp.num_elem++;
}
```

```
void rehash(TablaHash& tabla){
    TablaHash vieja = tabla;
    inicializarTabla(tabla,vieja.capacidad*2);
    nodo* tmp;
    for (int i=0; i<vieja.capacidad; i++){
        tmp = vieja.arr[i];
        while (tmp!=nullptr){
            insertar(tabla,tmp->clave,tmp->valor);
            tmp = tmp->proximo;
        }
    }
    vaciar(vieja);
}
```

Veamos en que consiste **el hashing cerrado (o direccionamiento abierto)** donde todos los elementos se almacenan en la tabla, no se crean listas

La **desventaja del hashing abierto** es que se debe mantener los objetos en listas y esto **consume tiempo**.


En el hashing cerrado todos los elementos permanecen en la tabla y en caso de **colisión** podemos intentar agregar el elemento en otra posición probando en las posiciones $h_0()$, $h_1(x)$, $h_2(x)$,... donde **$h_i(x) = ((\text{hashCode}(x) + f(i)) \bmod \text{Capacidad})$** con $f(0) = 0$. **Capacidad** un número primo.

Una primera estrategia de resolución de colisiones es **LINEAR PROBING**
(Sondeo lineal)

En este caso $h_i(x) = ((\text{hashCode}(x) + i) \% \text{TableSize})$

Veamos como va quedando la tabla al insertar 89, 18, 49, 58, 69 en una tabla con capacidad 10 y $\text{hashCode}(x) = x$

Inicialmente la tabla está vacía y insertamos 89, 18, 49, 58, 69 con $\text{hashCode}(x) = X$



	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Un objeto eliminado no puede ser realmente eliminado porque esto evitaría una búsqueda de otro elemento:

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

En la tabla si eliminamos 89 no podríamos luego buscar a 49, pues $\text{hash}(49) = 49 \bmod 10 = 9$ y estaría vacía esta celda y no podríamos seguir con el sondeo lineal.

Lo que hacemos es marcar la celda como no activa. **Tendremos entonces celdas activas, no activas y vacías**

Marcando las celdas podemos conseguir a 49 después de eliminar a 89:

0	49	a
1	58	a
2	69	a
3		v
4		v
5		v
6		v
7		v
8	18	a
9	89	n

Para buscar 49, aplicamos la función de hash, $49 \% 10 = 9$

La celda 9 es “no activa”, seguimos buscando en la celda $(9+1) \% 10 = 0$, y encontramos a 49

Luego si queremos insertar 29, como la celda 9 es “no activa”, lo podemos insertar allí.

Hagamos el ejercicio de insertar en la tabla con **sondeo lineal** los elementos **11, 21, 31, 34, 23, 33, 43** y eliminar **33** buscar **43**:

0		v
1		v
2		v
3		v
4		v
5		v
6		v
7		v
8		v
9		v

Hagamos el ejercicio de insertar en la tabla con **sondeo lineal** los elementos 11, 21, 31, 34, 23, 33, 43 y eliminar 33 y luego buscar 43:

Hagamos el ejercicio de insertar en la tabla con **sondeo lineal** los elementos **11, 21, 31, 34, 23, 33, 43** y eliminar **33**.

Respuesta:

0		v
1	11	a
2	21	a
3	31	a
4	34	a
5	23	a
6	33	n
7	43	a
8		v
9		v

Hay otro problema:

0	49	a
1	58	a
2	69	a
3		v
4		v
5		v
6		v
7		v
8	18	a
9	89	n

Si queremos insertar 29, como la celda 9 es “no activa”, lo podemos insertar allí. **Pero puede pasar que 29 esté en la tabla como “activo” (ya existe)**

Pues imaginen que al momento de insertar 89 luego insertamos 29, luego borramos 89 (se convierte en no activo), luego insertamos 29 en la casilla no activa que dejó 89. Y tendremos dos veces a 29 activo.

Podemos insertar 29 en la casilla no activa donde estaba 89 pero **hay que buscar antes a ver si 29 NO está como activo**

**Hagamos juntos insertar 2, 22, eliminar 2 e insertar 22 y
veamos el problema juntos**

0		v
1		v
2		v
3		v
4		v
5		v
6		v
7		v
8		v
9		v

insertar 2, 22, eliminar 2 e insertar 22

Insertar 2, 22, eliminar 2 e insertar 22 : como 22 está activo NO lo vuelve a insertar

0		v
1		v
2	2	n
3	22	a
4		v
5		v
6		v
7		v
8		v
9		v

Un **factor de carga recomendado** es que sea menor a **0.5** por lo que la tabla debe ser al menos el doble de tamaño del número de elementos en la tabla.

La **capacidad de la tabla un número primo** y una **función de hash que distribuya uniformemente los elementos del universo en la tabla**, es decir, el número promedio de elementos con igual posición en la tabla sea $|U|/TableSize$, donde $|U|$ es el número de elementos del universo de objetos.

No es evidente conseguir, pero hay resultados teóricos que permiten dar las recomendaciones vistas hasta ahora.

El principal problema del sondeo lineal es el **PRIMARY CLUSTERING** (agrupamiento primario), el cual consiste en que si un elemento a agregar cae en un grupo (cluster) necesitará varios sondeos (intentos) para ser insertado y su inserción además aumenta el grupo (cluster).

Al ir colocando en casillas contiguas aquellos elementos que tengan otra posición inicial pueden caer en el cluster

Veamos...

Ej: están llenas las casillas 1 2 3 4 después de haber insertado 11, 21, 31, 41

0		v
1	11	a
2	21	a
3	31	a
4	41	a
5		v
6		v
7		v
8		v
9		v

Si luego insertamos 63, en la casilla 3 está 31 y será insertado en la posición 5, aumenta el grupo (cluster)

Para evitar el **PRIMARY CLUSTERING** se utiliza el **SONDEO CUADRÁTICO**:

$$h_i(x) = (\text{hashCode}(x) + f(i)) \% \text{ TableSize}, f(0) = 0 \text{ y}$$

$f(i)$ es una función cuadrática. Normalmente se toma **$f(i) = i^2$**

Notemos que entre un sondeo y el siguiente hay una distancia fácil de calcular, por lo que no es necesario calcular i^2 cada vez:

$$h_i(x) - h_{i-1}(x) = i^2 - (i-1)^2 = 2*i - 1 \text{ para } i \geq 1$$

$$\text{Así } h_i(x) = h_{i-1}(x) + 2*i - 1$$

Ejemplo: Si $\text{hashCode}(x) = 1000$

$$h_0(x) = 1000, h_1(x) = 1000 + 1 = 1001, h_2(x) = 1001 + 3 = 1004,$$

$$h_3(x) = 1004 + 5, h_4(x) = 1009 + 4 \cdot 2 - 1 = 1009 + 7, \dots$$

Note también que $h_i(x) = h_{i-1}(x) + 2 \cdot i - 1 = h_{i-1}(x) + (2(i-1) - 1) + 2$



Entonces

$$h_i(x) = h_{i-1}(x) + \underbrace{2*i-1}_{\text{offset}(i)} = h_{i-1}(x) + \underbrace{(2(i-1)-1)}_{\text{offset}(i-1)} + 2$$

Así:

$$h_i(x) = h_{i-1}(x) + \text{offset}(i)$$

$$\text{offset}(i) = \text{offset}(i-1) + 2 \quad \text{y} \quad \text{offset}(1) = 1$$

Que usamos cuando programamos con sondeo cuadrático para reducir los cálculos

Veamos una función que aplica sondeo cuadrático para hallar la posición de un elemento x , y si no existe, devuelve una posición de una casilla vacía donde podemos, por ejemplo, insertar x

```
int encontrarPos( int x ) {// Aplicando sondeo cuadrático  $f(i) = i^2$ 
    // encontrar posición de x en la tabla o la posición en la tabla del último sondeo
    // si no está (que será una casilla vacía, donde podemos por ejemplo insertar a x).

    int offset = 1;
    int posActual = hashCode( x ); int cont_num_sondeos = 0;
    while ( Tabla.arr[ posActual ] != nullptr && Tabla.arr[ posActual ]->element != x &&
        cont_num_sondeos != Tabla.capacidad)
    { posActual = posActual + offset; // calcular siguiente sondeo  $h_i(x) = h_{i-1}(x) + 2i - 1$ 
      offset += 2; // se suma 2 al anterior sondeo
      if( posActual >= Tabla.capacidad) // si se pasa del rango aplicar la función mod de forma sencilla
          posActual - = array.length;
      cont_num_sondeos++; // si el while se ejecuta Tabla.capacidad veces NO hay casilla vacía
    }
    // se debería hacer rehash si devuelve -1 pues no consiguió casilla vacía ni donde esté x
    if (cont_num_sondeos != Tabla.capacidad) return posActual; else return -1;
}
```

Resultado teórico: Queremos insertar un elemento nuevo y la tabla tiene celdas vacías, para garantizar que encontraremos una celda vacía utilizando el **SONDEO CUADRÁTICO**, debemos tener:

- **TableSize un número primo**
- **Factor de carga menor a 0.5**

Por ejemplo si TableSize = 16 sólo las posiciones a distancia 0, 1, 4 y 9 pueden ser alcanzadas, pues se puede demostrar que $(i^2 \% 16)$ sólo puede valer 0, 1, 4 ó 9

Ejemplo:

Si $h_0(x) = 7$, $h_1(x) = 8$, $h_2(x) = 11$, $h_3(x) = 0$, $h_4(x) = 7$,

Ejemplo: Agregar 89, 18, 49, 58, 69 con Sondeo Cuadrático: sumar 1, 3, 5, 7 al anterior sondeo

$\text{hash}(X) = X \% 10$

Note que al
agregar 58,
prueba con
posiciones 8,
luego $8+1$ y luego
 $(8+4) \% 10 = 2$

Al agregar 69
prueba con las
posiciones $9, 9+1$
(0) y luego $9+4$ (3)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Hagamos juntos insertar con sondeo cuadrático 9 en la tabla (sumar 1 , 3, 5, 7 al anterior sondeo)

0	1	2	3	4	5	6	7	8	9
49		58	69					18	89

Hagamos juntos insertar con sondeo cuadrático 9 en la tabla (sumar 1 , 3, 5, 7 al anterior sondeo)

0	1	2	3	4	5	6	7	8	9
49		58	69					18	89

Resultado:

0	1	2	3	4	5	6	7	8	9
49		58	69		9			18	89

Aplicación:

Dado un arreglo `arr[]` de n números enteros y un valor dado,
Encontrar el número de pares de números enteros en el arreglo
cuya suma sea igual al valor objetivo.

Ejemplo: {1, 5, 1, 7, -1, 5, 5, 1} y 6

Resultado: 10

Veamos varios algoritmos...

Algoritmo basado en tabla hash:

Recorrer los elementos del vector desde el 0.

En cada iteración i , mantener en una tabla T cuántas veces aparece (objetivo-arr[i]) en el rango $i=0$ a $(i-1)$ y sumarlo al contador de número de pares.

Estando en la iteración i , para la siguiente iteración, las veces que aparece (objetivo-arr[i]) en el rango $i=0$ a i , se actualiza al sumar 1 a las veces que aparece (objetivo-arr[i]) en la tabla T

Hagamos juntos un ejemplo con el arreglo
1, 5, 5, 1, 7, -1, 1, 5, 1, 5 y objetivo = 6

1, 5, 5, 1, 7, -1, 1, 5, 1, 5 y objetivo = 6

```
// Programa C++ para contar el numero de pares que sumen  
// un valor dado usando Hash Map de C++  
// orden lineal (en promedio)
```

```
#include <iostream>  
#include <vector>  
#include <unordered_map>  
using namespace std;
```

```
// Devuelve el numero de pares en arr[0...n-1] con sum
// igual a "objetivo"
int contarPares(vector<int>& arr, int objetivo) {
    unordered_map<int, int> freq;
    int cnt = 0;

    for (int i = 0; i < arr.size(); i++) {

        // verifica si el complemento (target - arr[i])
        // existe en el map. Si es así, incrementa cnt
        if (freq.find(objetivo - arr[i]) != freq.end())
            cnt += freq[objetivo - arr[i]];

        // Incrementar la frecuencia de arr[i]
        freq[arr[i]]++;
    }
}
```

```
int main() {  
    vector<int> arr = {1, 5, 1, 7, -1, 5, 5, 1};  
    int objetivo = 6;  
    cout << contarPares(arr, objetivo);  
    return 0;  
}
```

PREGUNTAS???

