Proyecto 2: Hashing - Trees

Miguel Ochoa Hernández, Cesar Fernando Laguna Ambriz, Eduardo Armando Villarreal García
Maestria: Ciencias Computacionales
Universidad Autónoma de Guadalajara

***Resumen*- La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista. Este método, sin embargo, exige que los datos estén ordenados. Existen otros métodos que pueden aumentar la velocidad de búsqueda en el que los datos no necesitan estar ordenados, este método se conoce como transformación de claves (clavedirección) o hashing.**

I. Introducción

Para este tema lo siguiente es encontrar una forma de ordenar datos y acceder a ellos de tal manera que el tiempo sea rápido o casi instantáneo, para ello necesitamos primeramente hablar de una estructura de datos llamada Árbol.

Joyanes (p480) indica que El árbol es una estructura de datos fundamental en informática, muy utilizada en todos sus campos, porque se adapta a la representación natural de informaciones homogéneas organizadas y de una gran comodidad y rapidez de manipulación. Esta estructura se encuentra en todos los dominios (campos) de la informática, desde la pura algorítmica

(Métodos de clasificación y búsqueda...) a la compilación (árboles sintácticos para representar las expresiones o producciones posibles de un lenguaje) o incluso los dominios de la inteligencia artificial (árboles de juegos, árboles de Decisiones, de resolución, etc.).

Dentro de nuestro tema de estudio tenemos el tema de hashes lo cual es una forma trascendental para el tema de las inserciones y las búsquedas, en específico cuando tenemos una tabla hash.

Silva (p1) refiere que La tabla de hash pertenece a la categoría de diccionarios que son aquellas estructuras de datos y algoritmos que permiten buscar, insertar y descartar elementos.

Si las operaciones se reducen solamente a buscar e insertar se llaman tablas de símbolos.

En diccionarios puros sólo se implementa buscar.

II. Antecedente Algoritmo

Teniendo la premisa de que es una tabla hash ahora nos enfrentamos a las colisiones.

En informática, una colisión de hash es una situación que se produce cuando dos entradas distintas a una función de hash producen la misma salida.

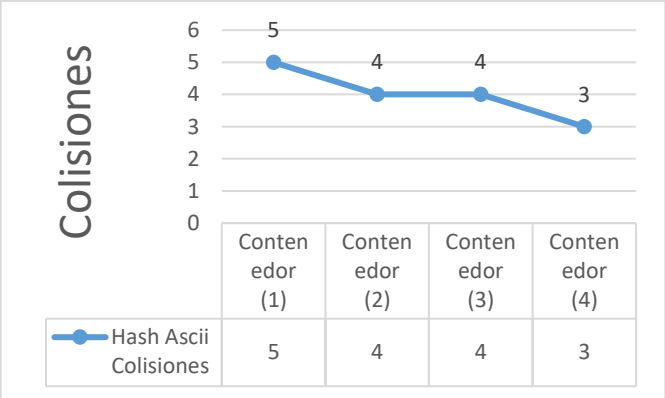
Es matemáticamente imposible que una función de hash carezca de colisiones, ya que el número potencial de posibles entradas es mayor que el número de salidas que puede producir un hash. Sin embargo, las colisiones se producen más frecuentemente en los malos algoritmos. En ciertas aplicaciones especializadas con un relativamente pequeño número de entradas que son conocidas de antemano es posible construir una función de hash perfecta, que se asegura que todas las entradas tengan una salida diferente. Pero en una función en la cual se puede introducir datos de longitud arbitraria y que devuelve un hash de tamaño fijo (como MD5), siempre habrá colisiones, debido a que un hash dado puede pertenecer a un infinito número de entradas.

III. Propuesta de Solución

A. Se aplicaron dos funciones hash implementadas de esta manera:

La primera función toma cada carácter de la cadena a ingresar en este caso el nombre y suma su valor que le pertenece en el código ASCIII obteniéndolo de esta forma:

```
return sum(ord(x) for x in string))
```



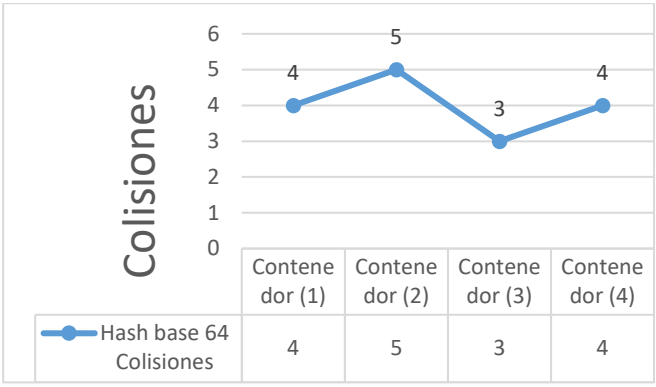
La segunda función en esencia es parecida solo que hace una conversión base 64 de cada carácter teniendo la función de esta manera:

```
for i in str(string):
    bits = bin(ord(i))[2:]
    bsize = len(bits)
    if bsize < 8:
        bits = "0" * (8 - bsize) + bits
    bstring = bstring + bits

encode = ""
while len(bstring) > 0:
    encode += __values[int(bstring[0:6], 2)]
    bstring = bstring[6:]

hashvalue = sum(ord(x) for x in encode)
```

Para este caso las colisiones encontradas fueron las siguientes:



B. **Aplicación para un Directorio Telefónico con las siguientes características:**

Los datos deben contener como mínimo los siguientes campos:

- Nombres
- Apellidos
- Dirección
- Celular
- Correo electrónico
- Redes Sociales

Para el desarrollo de esta aplicación se implementó un árbol rojo negro.

Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es rojo o negro. En adelante, se dice que un nodo es rojo o negro haciendo referencia a dicho atributo.

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas (NULL) son negras.
- Todo nodo rojo debe tener dos nodos hijos negros.
- Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

La toma de decisión del porque utilizar un árbol rojo negro es la poca dificultad de implementación con respecto a los demás arboles binarios

Para dar tratamiento al tema de la búsqueda dentro del árbol se implementó de la siguiente manera

Se implementaron dos diccionarios auxiliares uno de ellos almacena el hash de búsqueda por apellido y otro el hash de búsqueda por número telefónico, haciendo referencia a la tabla hash de nombre la cual es la búsqueda principal y llave, teniendo así una búsqueda de grado:

$$\theta(\log(n)) \qquad logaritmica$$

Teniendo la tabla hash se utiliza cada diccionario auxiliar para almacenar el numero hash de ese registro a consultar obteniendo así las ocurrencias en una búsqueda si se llegaran a dar elementos repetidos esto en el caso de apellido y en el caso de los elementos para la búsqueda por teléfono será más sencillo ya que estos son únicos.

```
hashvalue = self.hashfunction(register.name)
isEmpty = False
    if len(self.container[hashvalue]) == 0:
        isEmpty = True
        self.container[hashvalue].insert(register)
        self.numbers[register.cellphone] = hashvalue
        if register.last_name not in self.last_names:
            self.last_names[register.last_name]=
            [hashvalue]
        else:

         self.last_names[register.last_name].append(hashval
         ue)

         if not isEmpty:
             self.collisions[hashvalue] += 1
```

IV. Conclusiones

Podemos concluir que a medida que se requieren buscar datos con rapidez necesitamos la implementación de un hash la cual en primera instancia podría no ser del todo una solución completa dado las colisiones existentes al realizar inserciones, en todo caso se necesita dar tratamiento a esas colisiones con distintos métodos las cuales no se implementaron en esta problemática dada la naturaleza de la misma.

Referencias

[1] Joyanes., "Fundamentos de Programación", 2008, MacGraw Hill.

[2] Leopoldo Silva, "Tablas de Hash" (Abril 2008), Universidad Tecnica Federico Santa Maria. http://www2.elo.utfsm.cl/~lsb/elo320/clases/c7.pdf