Tecnológico de Monterrey
Edgar Cruz Vázquez – A01730577

TC1031.11: Programación de estructuras de datos y algoritmos fundamentales

Act 5.1 – Implementación individual de operaciones sobre conjuntos

Hashing:

Los hash o funciones de resumen se trata de algoritmos que consiguen crear a partir de una entrada (ya sea texto, contraseña, archivo, etc.) una salida alfanumérica de longitud normalmente fija que representa un resumen de toda la información que se le ha dado (crear a partir de los datos de la entrada crea una cadena que solo puede volverse a crear con esos mismos datos).

Estas funciones uno de sus varios cometidos es asegurar que no se ha modificado un archivo en una transmisión, hacer ilegible una contraseña o firmar digitalmente un documento.

Características:

Las funciones hash se encargan de representar de forma compacta un archivo o conjunto de datos que normalmente es de mayor tamaño que el hash independientemente del propósito de su uso.

Este sistema de criptografía implementa algoritmos que aseguran que con el hash no se podrá saber los datos insertados, indicando que es una función unidireccional. Tomando en cuenta que se pueden generar cualquier resumen a partir de cualquier dato, podría ocurrir el caso de que se podrían repetir estor resúmenes, a esto se le conoce como colisiones, pero esto no supone un problema ya que existen maneras para poder solucionar este tipo de casos usando técnicas para tener un buen algoritmo.

Ejemplo y formas de uso:

Hoy en día las funciones hash son muy usadas, una de las utilidades como ya se dijo con anterioridad es el proteger la confidencialidad de una contraseña, ya que podría estar en texto plano y ser accesible por cualquiera y aun así no poder ser capaces de deducirla.

En la comprobación de integridad de ficheros y también en cuestión para firmas digitales.

Manejo de colisiones:

Las tablas hash tratan las colisiones de dos formas.

Opción 1: Hacer que cada depósito contenga una lista vinculada de elementos que se han agregado a ese depósito. Esta es la razón por la que una función hash incorrecta puede hacer que las búsquedas en tablas hash sean muy lentas.

Opción 2: Si las entradas de la tabla hash están todas llenas, la tabla hash puede aumentar el número de depósitos que tiene y luego redistribuir todos los elementos de la tabla. La función hash devuelve un número entero y la tabla hash tiene que tomar el resultado de la función hash y modificarlo contra el tamaño de la tabla de esa manera puede estar seguro de que llegará al cubo. Entonces, al aumentar el tamaño, repetirá y ejecutará los cálculos de módulo que, si tiene suerte, podrían enviar los objetos a diferentes cubos.

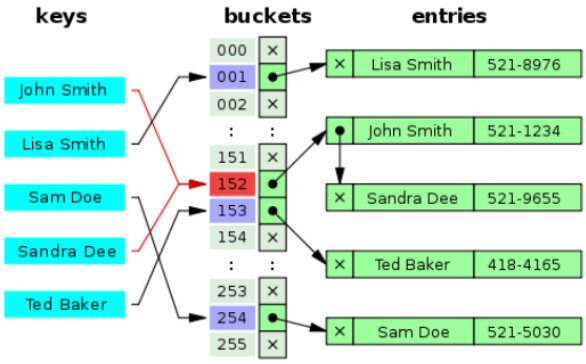
Tecnológico de Monterrey
Edgar Cruz Vázquez – A01730577

TC1031.11: Programación de estructuras de datos y algoritmos fundamentales

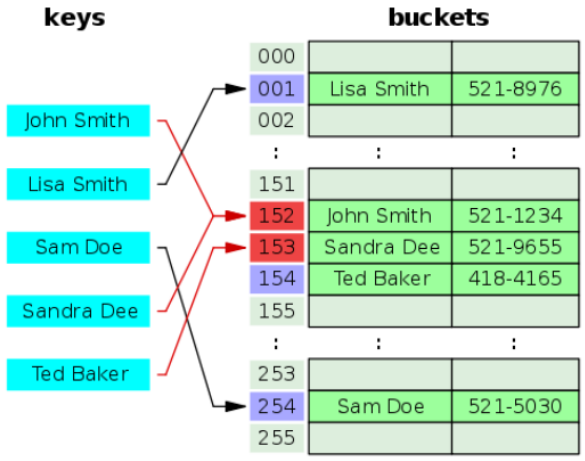
Act 5.1 – Implementación individual de operaciones sobre conjuntos

Explicación:

- **Encadenamiento separado**



- **Direccionamiento abierto**



Complejidad:

Las tablas hash normalmente tienen un timpo de complejidad para las funciones (Búesqueda, inserción y eliminación) de O(1), siendo la mejor de todas y en sus peores casos de O(n) que tampoco es malo, en cuanto al espacio de complejidad en su peor forma tiende a ser O(n log(n)) pero entonces ¿por qué necesitamos usar otras estructuras de datos ya que las operaciones de hash son tan rápidas? ¿Por qué no podemos simplemente usar tablas hash / hash para todo?

- El consumo de memoria puede ser mucho mayor. Esto es cierto si se utilizan tablas hash para reemplazar matrices.
- Hay algunas operaciones que no son compatibles de manera eficiente con las tablas hash, como iterar sobre todos los elementos cuyas claves están dentro de un cierto rango, encontrar el elemento con la clave más grande o la clave más pequeña.
- Las tablas hash no están ordenadas
- Las tablas hash no son las mejores para insertar cabeza / cola.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Tecnológico de Monterrey
Edgar Cruz Vázquez – A01730577

TC1031.11: Programación de estructuras de datos y algoritmos fundamentales

Act 5.1 – Implementación individual de operaciones sobre conjuntos

Para la creación del código se tomo como referencia el código de la siguiente pagína: [link], se hicieron pequeños cambios a conveniencia para el ejercicio, la muestra del código se hizo con propósito educativo y sin tratar de demeritar o causar plagio con el creador.

Chain():

Esta función recibe como parámetros la tabla hash, el array de los datos a insertar, y un entero que es el tamaño de dicho arrray de los datos, en donde con un ciclo for este ira iterando por el array de los datos y en cada iteración entra a una función en la cual se llama insertItem(), en donde recibe el dato a insertar a la tabla y dentro de la función se obtiene el índice en el cual se insertara a la tabla mediante modulación, en caso de que pueda existir alguna colisión en donde se llegue a repetir la llave para la indexación, se creara una linked list dentro del índice guardando los datos que obtengan dicho índice:

```
void Hash::chain(Hash h, int a[], int n){
    for (int i = 0; i < n; i++)
    h.insertItem(a[i]);
}
```

```
void Hash::insertItem(int key)
{
    int index = hashFunction(key);
    table[index].push_back(key);
}
```

```
int Hash::hashFunction(int x){
    return (x % BUCKET);
}
```

Tecnológico de Monterrey
Edgar Cruz Vázquez – A01730577

TC1031.11: Programación de estructuras de datos y algoritmos fundamentales

Act 5.1 – Implementación individual de operaciones sobre conjuntos

Quadratic():

En la función quadratic este recibe como parámetros la tabla hash, su tamaño, el array de los datos a insertar junto con también el tamaño de dicho array, se hace un ciclo for para pasar por el arreglo de los datos a insertar, estando en cada iteración la creación de la llave siendo el índice en el cual se insertar a la tabla por medio de modulación, pero en caso llegue a existir una colisión se hará otra iteración for en donde pasara por toda la tabla hash y en cada iteración se estará creando una nueva llave con el método de quadratic probing, y verificando que con la nueva llave siendo este el nuevo índice que este desocupado el espacio y en caso este vació se insertara el dato.

```
// Function to implement the
// quadratic probing
void quadratic(int table[], int tsize,
               int arr[], int N)
{
    // Iterating through the array
    for (int i = 0; i < N; i++)
    {
        // Computing the hash value
        int hv = arr[i] % tsize;

        // Insert in the table if there
        // is no collision
        if (table[hv] == -1)
            table[hv] = arr[i];
        else
        {
            // If there is a collision
            // iterating through all
            // possible quadratic values
            for (int j = 0; j < tsize; j++)
            {
                // Computing the new hash value
                int t = (hv + j * j) % tsize;
                if (table[t] == -1)
                {
                    // Break the loop after
                    // inserting the value
                    // in the table
                    table[t] = arr[i];
                    break;
                }
            }
        }
    }
}
```

Impresión:

```
Insercion tipo chain.
0
1 --> 15 --> 8
2
3
4 --> 11
5 --> 12
6 --> 27

Insercion por metodo cuadratico.
700 50 85 73 101 92 76
```

Conclusión:

Para este trabajo se hizo la introducción de datos para para la seguridad de datos de los usuarios. Se analiza sus características y en qué casos es factible el uso de estos en ciertas situaciones, la complejidad que maneja siendo estas de las mejores que existen, pero también teniendo sus desventajas para la realización de ciertas funciones o propósitos, este tema me hizo ver los diferentes métodos para inserción de datos y la seguridad de estos, entender que los algoritmos de HASH no es como tal un algoritmo de encriptación, aunque si se utiliza en esquemas de cifrado y que tiene varios usos como la comprobación de la integridad de ficheros, seguridad en procesos de identificación en sistemas, firma digital, entre otros.